AFRL-IF-WP-TR-2001-1549

COMPUTER AIDED ENGINEERING FOR RECONFIGURABLE COMPUTING (CAERC)

DR. RANGA VEMURI

UNIVERSITY OF CINCINNATI
LABORATORY FOR DIGITAL DESIGN ENVIRONMENTS
DEPARTMENT OF ECECS, ML. 30
CINCINNATI, OH 45221-0030

OCTOBER 2001

FINAL REPORT FOR PERIOD 25 JULY 1997 – 30 SEPTEMBER 2001

Approved for public release; distribution unlimited

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

Al Scarpelli

AL SCARPELLI
Project Engineer/Team Leader
Embedded Info Sys Engineering Branch
Information Technology Division

James S. Williamson

JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
October 2001	Final	07/25/1997 - 09/30/2001

4. TITLE AND SUBTITLE
COMPUTER AIDED ENGINEERING FOR RECONFIGURABLE COMPUTING (CAERC)

5a. CONTRACT NUMBER
F33615-97-C-1043

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62204F

6. AUTHOR(S)
DR. RANGA VEMURI

5d. PROJECT NUMBER
6096

5e. TASK NUMBER
40

5f. WORK UNIT NUMBER
37

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
UNIVERSITY OF CINCINNATI
LABORATORY FOR DIGITAL DESIGN ENVIRONMENTS
DEPARTMENT OF ECECS, ML. 30
CINCINNATI, OH 45221-0030

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2001-1549

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT *(Maximum 200 Words)*
Synthesis and Partitioning for Adaptive and Reconfigurable Computing Systems (SPARCS) is a computer aided engineering environment for reconfigurable computers. SPARCS contains software implementations of a variety of methods and algorithms for various subproblems for automating the task of producing designs for multi-FPGA (Field Programmable Gate Array) based reconfigurable computers. The SPARCS system includes tools for temporal partitioning, spatial partitioning, high-level synthesis, physical design, and arbiter synthesis. This is a comprehensive report on the SPARCS project, describing various techniques developed for solving these problems. In addition, this report contains some experimental results demonstrating the effectiveness of the SPARCS tools.

15. SUBJECT TERMS:
Reconfigurable Computing, Computer Aided Design, Design Automation, Field Programmable Gate Arrays, Partitioning, Synthesis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Alfred Scarpelli
Unclassified	Unclassified	Unclassified	SAR	186	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-6548 x3603

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Goals and Objectives

Reconfigurable processors consisting of a sea of uncommitted FPGAs offer the same performance advantages of custom computing while retaining the flexibility of general purpose instruction architectures. In 1996, at the begining of this project, it was predicted that in near future reconfigurable processors can offer 100x performance improvement over contemporary microprocessors, and 10-100x reduction in power/gate, 20x progress in density, and 1,000,000x reduction in reconfiguration time compared to current reconfigurable devices.

Unfortunately, state-of-the-art design synthesis methodologies were able to use perhaps 50-75% of the available gates at 30-50% of the maximum clocking rate of single reprogrammable device and were woefully inadequate in synthesizing multi-device systems. In order to deliver the performance expectations of reconfigurable architectures, dramatic improvements in the synthesis tools were necessary in both directions: (1) ability to synthesize to multiple device architectures, and, (2) improvement in utilization and delivered performance of each device.

The goal of this project was to develop a coherent design synthesis and partitioning environment for complete, automated synthesis for reconfigurable processors.

More specifically, the following goals were established for this project.

- Development of new architecture-driven algorithms for both temporal and spatial partitioning of applications for reconfigurable processors.
- Development of a tool framework for fully coordinated partitioning-synthesis process and integration of existing as well as new tools into this framework.
- Development of new performance-driven FPGA floor-planning algorithms and new FPGA pin-assignment algorithms for efficient physical design of reconfigurable processors.
- Incorporation of physical synthesis algorithms into behavioral synthesis in order to generate highly accurate resource and performance estimates during synthesis.
- Demonstration of the synthesis environment in conjunction with a low-cost, portable, reconfigurable processor specifically developed for this program.

Based on the Air Force Research Laboratory's objectives stated in the PRDA announcement and considering pragmatics of the state-of-the-art in the area we conceived this program with the following primary objectives of equal importance:

- *This program shall enhance the state-of-the-art multi-FPGA synthesis tools, developing new algorithms where necessary as identified in this proposal, and integrate them into a coherent design synthesis environment for complete, automated synthesis for reconfigurable processors.*
- *This program shall demonstrate complete, coordinated synthesis capability, combining both university and industry tools, for delivering the intrinsic capacity of reconfigurable devices outside the device boundaries, at the system reconfigurable system level.*

Unique contributions of this program include,

- Specific targeting of all the tools to *selectively reconfigurable architectures* which are most suitable for field-deployment in defense applications and which, as explained in the previous section, accommodate both static and dynamic reconfigurability.
- Development of a architecture independent synthesis environment where all of tools accept the architecture specification of the reconfigurable processor as an explicit input.
- Development of a tool framework for fully coordinated partitioning-synthesis process and integration of existing as well as new tools into this framework.
- Development of new architecture-driven partitioning algorithms so as to make the synthesis-partitioning process independent of the specific architecture of the reconfigurable processor.
- Development of new performance-driven FPGA floor-planning and placement algorithms and new FPGA pin-assignment algorithms for efficient physical design of reconfigurable processors.
- Incorporation of physical synthesis algorithms into behavioral synthesis in order to generate highly accurate resource and and performance estimates during synthesis.
- Demonstration of the synthesis environment in conjunction with a low-cost, field-deployable, selectively reprogrammable multi-FPGA architecture, specifically developed for this program for a selected avionics application.

1.2 Summary of Technical Issues

Following is a summary of key technical issues addressed in the proposed program.

1. *Architecture specification*

   Our goal is to develop a synthesis and partitioning environment that is independent of the specific architecture of the target reconfigurable processor. Key to this is the ability to

specify the key components of the architecture in way the tools can make use of them. This task deals with the development of a notation for the specification of the target architecture to the various tools in our environment. The notation will be VHDL-based and will make use of the component declaration and configuration facilities in VHDL.

2. *Architecture-Driven Partitioning*

   Partitioning continues to be a key element for any multi-FPGA system. In case of reconfigurable processors partitioning has two dimensions:

   - *Temporal Partitioning:*

     Temporal partitioning deals with the partitioning of a specification into a number of ordered subtasks so that the processor needs to be reconfigured between tasks. Recall that the selectively reconfigurable architectures offer unlimited resources only a finite number of which can be used at any time. The temporal partitioning step divides the specification so that each specification segment can be mapped to the processor by suitably configuring the FPGAs available. Primary issues to consider in temporal partitioning include reconfiguration costs including the temporary memory needed for live data storage (context) and reconfiguration time including the time to save and restore the context.

   - *Spatial Partitioning:*

     Spatial partitioning deals with a more traditional task of partitioning a specification subtask onto the multiple FPGA resource available in the processor. This is the step where the basic reconfiguration sketch of the processor is determined.

   The overall partitioning involves coordination between the temporal and spatial partitioning steps. For a given specification, at most one temporal partitioning resulting in a number of subtasks is necessary. For each temporal specification segment, a spatial partitioning is necessary to determine optimal mapping of the subtask on the FPGAs and hence optimal configuration.

   Temporal partitioning yields a *reconfiguration schedule* that determines when the processor will be reconfigured. Spatial partitioning yields a specific reconfiguration bit-streams for each reconfiguration step in the reconfiguration schedule. Note that although the reconfiguration schedule is statically determined, it is carried-out dynamically during the application run-time. Note that both temporal and spatial partitioning should be driven by the target architecture specification.

3. *Macro Based Floor Planning and Pin-Assignment*

   Two important issues to address during physical design include floor planning and pin assignment. As discussed in the previous section, lack of proper floor-planning considering the macro-cell structures leads to poor utilization and poor performance. Similarly, lack of proper signal to pin assignments at the board level results in poor board-level performance. The pin assignment problem is complicated by the existence of certain pre-determined pins and signal locations on reconfigurable boards.

4. *High Level and Layout Synthesis Integration*

   As noted before, to obtain high-performance designs from high-level synthesis systems, physical design automation tools must be integrated within the high-level synthesis flow.

The key impediment to this is the time consumed for each physical synthesis iteration during the high-level synthesis process when a large number of alternative structures are considered. Our approach involves complete integration of high level and layout synthesis algorithms, by developing light weight versions of layout synthesis algorithms to be integrated into the high level synthesis tool.

5. *Partitioning-Synthesis Interaction*

   The question often asked is "Is partitioning first or synthesis first?". Our studies showed that there is no answer that fits all specifications. Some (small to medium scale) specifications can be more effectively partitioned at the RTL/gate levels whereas some (relatively large scale) specifications can be more effectively partitioned at the behavior level. It is however clear that regardless of when partitioning is done, it should effectively interact with the subsequent synthesis step and vice-versa. It is best to view partitioning and synthesis as integrated within the single task of evolving the best *hierarchical* structure.

1.3 Description of Tasks

The following specific tasks constituted the statement of work:

1. RC Architecture Specification Style

   Develop a style for specification of selectively programmable reconfigurable computer architectures.

2. Temporal Partitioning

   Develop temporal partitioning algorithms for VHDL specifications of reconfigurable computer applications.

3. Architecture-Driven Spatial Partitioning

   Develop architecture-driven spatial partitioning algorithms, including pin-assignment, for reconfigurable computer applications.

4. Partitioning with Synthesis

   Integrate the spatial and temporal partitioning algorithms with light-weight synthesis algorithms, at behavioral, RTL and gate levels.

5. High-level Synthesis with Physical DA

   Integrate high-level synthesis algorithms with light-weight layout synthesis algorithms.

6. Light-weight versions of Existing Synthesis Algorithms

   Develop light-weight versions of existing synthesis algorithms (behavioral, layout and logic) suitable for embedding within other synthesis/partitioning programs.

7. RC and FPGA Floor-Planning

   Develop algorithms for single and multiple FPGA floor planning taking performance considerations into account.

8. Portable RC Development for Demonstration

   Develop the portable reconfigurable computer system hardware for demonstration purposes.

9. Prototype Software Development, Testing and Documentation

   All algorithms, techniques and tools shall be implemented on standard Unix workstations, primarily in C and C++.

1.4 Overview of the Report

This is a comprehensive final technical report on the "Computer Aided Engineering for Reconfigurable Computing (CAERC)" project. This report is divided into the following chapters:

1. *Reconfigurable Computing (RC) architecture Specification Style in VHDL*

   Chapter 2 describes the Unified Specification Model (USM) developed for specification of Reconfigurable Computing (RC) computations.

2. *Temporal Partitioning*

   Chapter 3 describes the research on temporal partitioning of an application for mapping to a reconfigurable architecture.

3. *Architecture-Driven Spatial Partitioning*

   Chapter 4 describes methods for architecture-driven spatial partitioning for reconfigurable computing applications.

4. *Partitioning with Synthesis*

   Chapter 5 describes methods for integrating partitioning with synthesis at the behavior level.

5. *Light Weight Versions of Existing Synthesis Algorithms*

   Chapter 6 describes light weight synthesis algorithms for use in performance estimation during partitioning.

6. *RC and FPGA Floor Planning*

   Chapter 7 describes floor planning techniques for reconfigurable architectures.

7. *Portable RC Development for Demonstration*

   Chapter 8 describes the Portable and Reconfigurable Computer (PARC) architecture.

8. *Prototype Software Development, Testing and Demonstration*

   Chapter 9 describes the SPARCS (Synthesis and Partitioning for Adaptive, Reconfigurable Computing Systems) system software. SPARCS is an implementation of the various algorithms and techniques developed in this research.

Chapter 2

RC Architecture Specification Style in VHDL

2.1 Introduction

This chapter describes the specification of an application in the RC framework[1]. The specification language used to model an application is very important since it molds the application in a way such that synthesis tools can process it. Furthermore, a good modeling language exposes many characteristics of the application: parallel threads of execution can be neatly presented, data structures readily formed, etc. In memory synthesis, the requirements of an efficient modeling language include being able to introduce address generation logic without intrusive modifications. Once data structures of the application are mapped to memory banks of the hardware, the process of adding address generation logic and arbitration mechanisms should be straightforward. Also, for memory synthesis, a succinct representation of the variables and data structures of an application is essential. In order to map data structures of the application onto banks of the RC, data structures should be extracted from the application and presented in a form suitable for synthesis.

For the memory assignment problem described in this chapter, a hierarchical representation of an application can capture the implicit parallelism of computational tasks in the design while hiding the flow of each task's control thread. Furthermore, this hierarchical representation successfully captures the relationship of computational tasks with the variables of the design.

The hierarchical representation presented in this chapter consists of two specification styles (an illustrative example is shown in Figure 2.1):

- *Fine-grain representation:* First, each computational task is represented in the Behavior Blocks Input Format (BBIF). This model, illustrated in Figure 2.1a is a hierarchical Control Data Flow Graphs (CDFG) representation that captures the behavior of computations. Section 2.2 further describes the BBIF format and Appendix A provides further formal definitions.

[1]This work was done in conjunction with the SPARCS [26] research team.

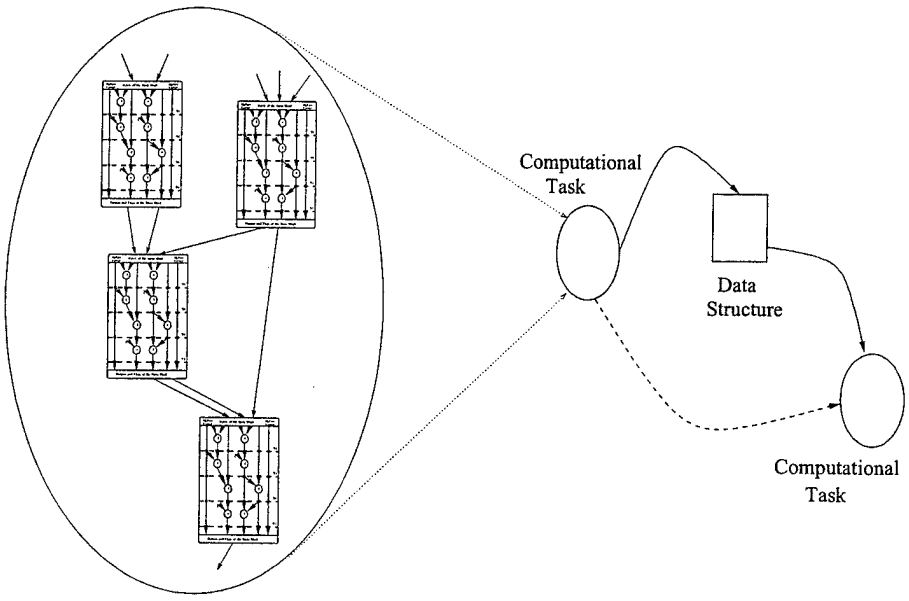


Figure 2.1: **Hierarchical Modeling**

- *Coarse-grain representation:* Second, the overall flow graph is represented in the Unified Specification Model (USM) format. The USM, illustrated in Figure 2.1b captures the concurrency and coordination model of a design in a style compatible with hardware description languages such as VHDL. Section 2.3 defines the USM format.

2.2 BBIF Specification

A computational model is needed to capture the behavior of the fine-grain level of parallelism in a design. The Behavior Blocks Input Format (BBIF)[49] is a hierarchical CDFG representation with features well-suited for High-Level Synthesis. A BBIF model represents a behavioral *task* with a single thread of control. The BBIF is organized as a list of behavior blocks, where the data flow and computations are captured within each behavior block, while the control flow is captured at the inter-block level. The task interacts with the environment through design input and output ports that are visible across all behavior blocks. The control flow starts at the first behavior block and transfers from one block to another through the branch construct provided at the end of each block. The `branch` statement specifies either an unconditional transfer to a single successor block or a conditional transfer to *one* of the series of successor blocks.

In conjunction with the USM specification presented in the next section, the BBIF specification provides a modeling environment in which inter-task as well as intra-task parallelism can be efficiently expressed.

The USM specification, presented in Section 2.3, plays an important role since the memory mapping problem is concerned with the inter-task interface and protocol. BBIF definitions and

notations are provided in Appendix A. A formal and more detailed description of the BBIF model is provided in [49].

2.3 USM Specification

2.3.1 Introduction

This section proposes a Unified Specification Model (USM) of concurrency and coordination compatible with VHDL. The specification model embodies a uniform treatment of computation, communication channels, and memories, facilitating its use across a variety of synthesis applications. We discuss synthesis semantics of the USM representation and the advantages of the USM synchronization model in comparison to similar VHDL motivated representations.

VHDL has been used for behavior level specifications for a variety of high-level synthesis tools, hardware-software co-synthesis systems, and adaptive system synthesis environments. VHDL provides a rich set of high-level constructs to permit succinct specification of concurrent and coordinating processes. A variety of intermediate representations [10, 75, 72, 41, 54, 2] have been proposed to capture various specification elements in VHDL in a form suitable for further processing during synthesis. These include data-flow graphs (DFG), mixed control-data flow graphs (CDFG), timed decision tables (TDT), and various flavors of graph-based and table-based formalisms sometimes augmented with global flow information such as module call graphs (MCG). Although many of these representations share common features, they also have application-specific features that inhibit their use across various types of synthesis systems.

This section presents an overview of the Unified Specification Model, and provides insight on compatibility with VHDL.

The USM representation can be used for: 1) high-level VLSI synthesis [32] where the goal is to synthesize a CMOS ASIC; 2) hardware-software co-synthesis [62] where the target architecture contains a general-purpose processor to implement software tasks and a coprocessor to implement the hardware tasks; and 3) adaptive system synthesis [26] where the target architecture is a dynamically reconfigurable multi-FPGA board with both local memories for each FPGA and a shared memory, and a crossbar type communication fabric.

In the next sections, we present a detailed description of the Unified Specification Model and its semantics.

2.3.2 Unified Specification Model

The Unified Specification Model is a hierarchical representation for specifying the behavior of a design. The designer can also specify the behavior of the environment in which the design is to execute. A USM example is shown in Figure 2.2. At the highest level of the USM are two types of objects called tasks and memory segments. Tasks in the USM represent elements of computation and memory segments represent elements of data storage. Tasks are classified into design tasks and environment tasks. Environment tasks are written primarily to specify the I/O model between the design tasks and the environment. Hence, design tasks are those that are synthesized and environment tasks are only used to extract information about the I/O interface and protocol.

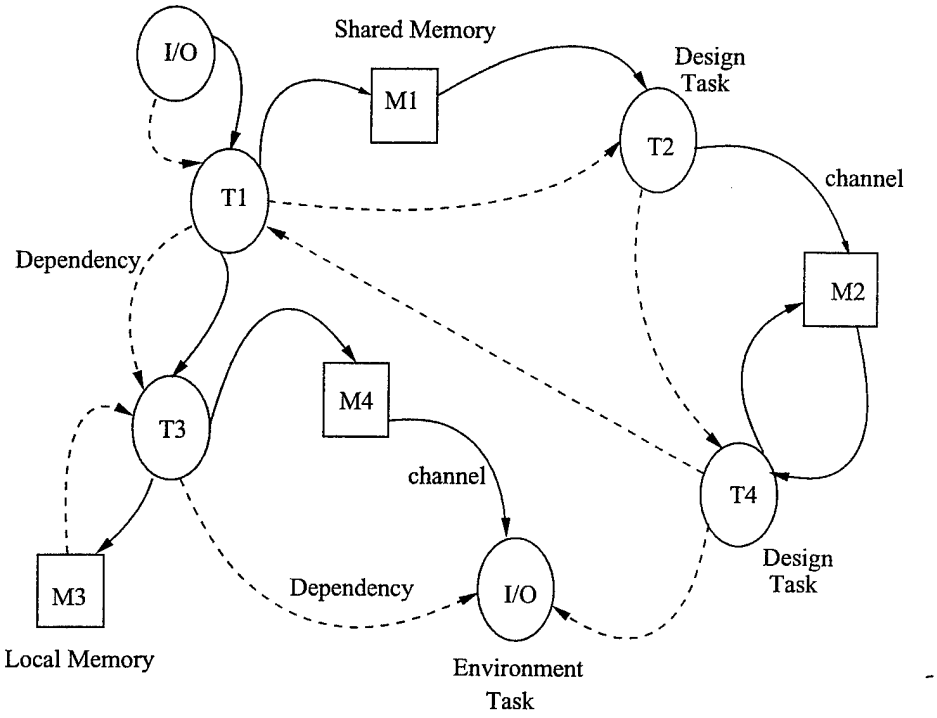


Figure 2.2: **USM Task Graph Example**

All tasks in the USM are simultaneously executing so as to model concurrency. USM objects (tasks and memory segments) can be connected through edges that are either channels or dependencies. USM allows the specification of dependencies among tasks in order to represent coordination. Channels are used to represent inter-task and task-to-memory communications.

In the following paragraphs, we will present the semantics of the memory segments, communication channels, dependencies, and finally present a detailed description of how the computation within a task is specified in the USM.

Memory segments

A memory segment is an element of storage whose size and word length are defined by the user. A memory segment can be declared as being local to a task (M3 in Figure 2.2), or shared between multiple tasks (M1 in Figure 2.2). Both environment and design tasks have access to memory segments.

When a memory segment is used by more than one task as a means of transferring data from one task to another (i.e. one task is writing to and another is reading from the memory segment), the designer is responsible for synchronizing the tasks using dependencies. However, if multiple tasks are accessing different areas of the same segment, or if the tasks are only reading from the segment, no synchronization is required. It is assumed that the synthesis process would introduce memory arbitration between the tasks whenever needed. This way, the designer need not worry

about resolving conflicts between memory access operations. This keeps the design architecture-independent: the synthesis process can map the memory segment to a physical memory that has either one port or multiple ports; Also, the synthesis process can map multiple memory segments to the same physical memory. Memory segments are easily implemented in VHDL as local or shared variable arrays.

Communication channels

Channels provide the means of communication between tasks (environment as well as design tasks), and between tasks and memory segments. Depending on the bitwidth required between source and destination, the designer must fix the size of the channel. Between each pair of communicating objects, one or more channels could be used. However, the design should not share the same channel across different pairs of objects. Again, if the synthesis process decides to share channels due to resource constraints, then it will automatically identify those channels and provide arbitration. This simplifies the task of the designer since manual introduction of arbitration mechanism for shared channels is not required.

A communication channel used by the designer is unidirectional. However, since synthesis tools might introduce channel sharing, a physical channel might be bi-directional.

In VHDL, when implementing USM channels, signals with the appropriate bitwidths can be used.

Dependencies

Dependencies are used as means of providing explicit synchronization for data and control flow between tasks. A dependency is a directed control line from a source task to a destination task. The semantics of a dependency edge implies that a destination task waits until its corresponding source tasks initiate its execution. Source and destination tasks can be either environment or design tasks. There can be multiple destination tasks dependent on a source task through a single dependency edge. However, a task may be dependent on several tasks through separate dependency edges. The flow of task execution is captured by the dependencies in the task graph. More accurately, the role of a dependency edge is two-fold: it provides synchronization between one-time executing tasks, and provides synchronization mechanism between tasks involved in loops. The following two paragraphs present the semantics of these dependencies.

Synchronization using dependencies: Dependency edges provide a way of synchronizing task execution. This is equivalent to synchronizing data transfer from one task to the other. A 1-bit control flag is associated with each dependency edge. This synchronization mechanism allows tasks to start execution irrespective of the status of other tasks executing in parallel.

A dependency edge used to order execution of tasks is shown in Figure 2.3 and a dependency edge used to synchronize data transfer is shown in Figure 2.4. In the example shown in Figure 2.3, task T2 waits for the completion of task T1 before it begins execution. Whereas in Figure 2.4, task T2 waits for task T1 to write a value into the channel c12. In these figures, notice that the constructs Raise()/IsRaised() are used to set/check the value of the control flag. In order to synchronize two tasks, the source task invokes the Raised() construct on a flag, whereas the destination task waits in a loop invoking the IsRaised() construct on the same flag.

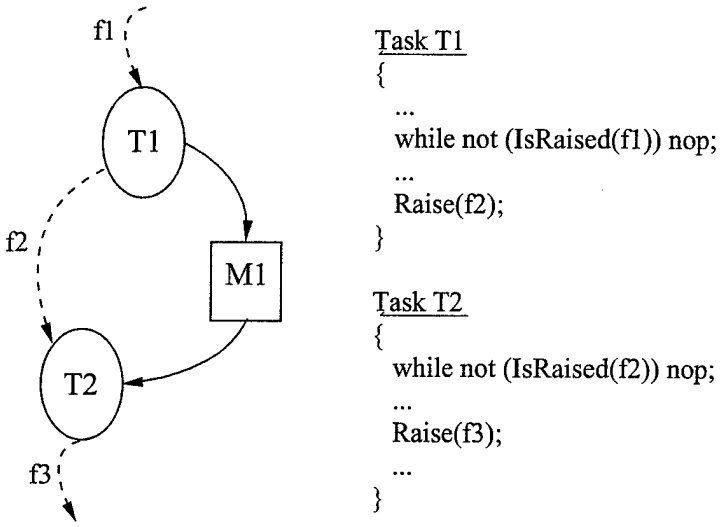


Figure 2.3: **Synchronization of Task Execution**

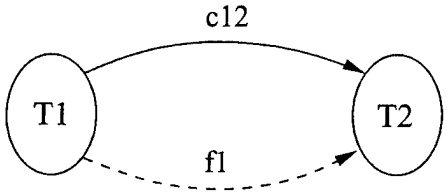
This synchronization mechanism ensures that the destination task waits until the source task triggers its execution or that the data is ready for consumption. Note that this type of dependency edge also allows the conditional execution of tasks. For example in Figure 2.5, the source task T1 raises either flag f12 or f13 based on a condition. Therefore, tasks T2 and T3 are conditionally dependent on T1.

Conditional dependencies imply that a destination task need not finish execution but may wait indefinitely. Hence, the execution semantics of the USM is defined as follows: The execution cycle for a collection of tasks is defined to finish when all the tasks are indefinitely waiting. The model assumes that there is an indefinite wait at the end of each task.

In VHDL process synchronization, all tasks have to arrive at a wait state before waiting tasks can be triggered. However, in USM, the triggering mechanism is not based on the wait command, instead, each pair of tasks has its own busy-wait synchronization procedure. This allows multiple processes to run concurrently without the need for global synchronization. On the other hand, this busy-wait procedure can be easily implemented in VHDL. For simulation purposes, a busy-wait can be replaced by a simple VHDL wait without loss of functionality.

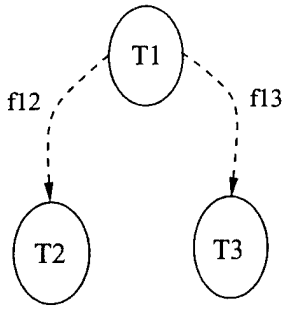
Loops using dependencies: Dependency edges are also used to implement loops in a task graph. Since tasks involved in loops might execute more than once, a mechanism is needed to ensure proper inter-task synchronization.

The (Raised(), IsRaised()) mechanism explained in Section 2.3.1 is not adequate to represent a loop dependency. This is because there is no control on the number of times a destination task might execute: Once the flag it is waiting on is raised, a destination task might start executing its first iteration and then incorrectly proceed to a subsequent iteration since the flag might still be raised. A solution to this problem is to provide a dependency based on a flag that is toggled each time the source needs to trigger.




```
Task T1                          Task T2
{                                {
  ...                              while not (IsRaised(f1)) nop;
  c12 <= x;                        y <= c12;
  Raise(f1);                       ...
  ...                            }
}
```

Figure 2.4: **Synchronization of Data Transfer**

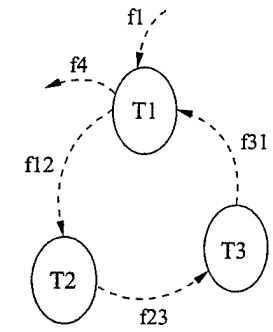



```
Task T1
{
  ...
  if (...) then Raise(f12);
  else Raise (f13);
  ...
}
```

```
Task T2
{
  while not (IsRaised(f12)) nop;
  ... body of T2 ...
}
```

```
Task T3
{
  while not (IsRaised(f13)) nop;
  ... body of T3 ...
}
```

Figure 2.5: **Conditional Task Execution**




```
Task T1
{
  for (...) do
  {
    ...
    f12 = not(f12);
    ...
    while (f31 == old_f31) nop
    old_f31 = not(old_f31);
  }
  ...
  Raise(f4);
}
```

```
Task T2

  loop forever
  {
    ...
    while (f12 == old_f12) nop;
    old_f12 = not(old_f12);
    ...
    f23 = not(f23);
    ...
  }
}
```

```
Task T3
{
  loop forever
  {
    ...
    while (f23 == old_f23) nop;
    old_f23 = not(old_f23);
    ...
    f31 = not(f31);
    ...
  }
}
```

Figure 2.6: **Loop Dependencies**

Hence, to synchronize two tasks involved in a loop, a toggling dependency edge is used. Each destination task should not only wait on the value of the flag but also on the event on the flag. Thus, it is imperative for the destination task to keep track of the old value of the control flag. This is advantageous since only a single bit value has to be passed between the source and the destination tasks.

As a result, instead of using two dependency edges (with the Raise/IsRaised mechanism) to ensure proper execution of loops, a toggling dependency edge solves the problem by introducing only one flag (instead of two) and a local storage bit in the destination task.

Figure 2.6 shows an example of a loop involving three tasks: T1, T2, and T3. The control flows into T1 through the dependency f1. T1 triggers T2 for a number of times specified by the for loop inside T1, after which it raises flag f4 and stops. Task T2 is in turn dependent on T1 and will only be triggered by flag f12 originating in T1. Similarly, T3 is dependent on T2 through flag f23. Finally, T3 triggers the next iteration of T1 through flag f31.

Clearly, for a dependency edge involved in a loop, one 1-bit control flag is still needed but an additional 1-bit storage in each destination task is required. Note that, initially, the value of the control flag and all corresponding 1-bit storage flags should be the same (either 0 or 1).

Tasks

A Task (shown in Figure 2.7) consists of a set of inputs and outputs which can be flags, shared memories and channels, some of which representing design I/O, and a set of local storage

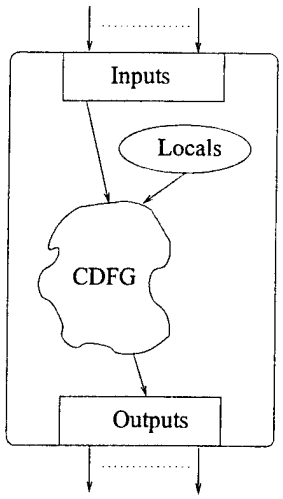


Figure 2.7: **Task Model**

elements that are variables/constants, local memories, or flags. Flags, as mentioned before, are special storage elements used to represent inter-task dependencies.

Computations within a task are represented as a Control Data Flow Graph (CDFG). The CDFG is a directed acyclic graph where the nodes represent operations and the edges represent data/control flow. A node in the CDFG can be represented by a 4-tuple given as:

$$< name, input\_set, output\_set, data\_dependency\_set >$$

The name denotes the ID of the operation, the input_set is the set of all storage elements input to the node, the output_set is the set of all storage elements output from the node, and the data_dependency_set consists of the set of all source nodes that a node depends on.

The entire CDFG can be built in a two-step process. A flow graph from the source language can be first generated and, after a detailed dependency analysis, a dependency graph can be generated. In the case of a flow graph, the input and output sets of the nodes are formed and the dependency sets of the nodes are empty.

An example flow graph and a corresponding input specification are shown in Figure 2.8. The edges in the graph are annotated by the storage names that they represent.

The nodes in a CDFG can be broadly classified into these two types:

- *Control Nodes:* These nodes are used to represent the flow of control and the CDFG is organized as a collection of control nodes connected by control edges. Conditional constructs such as if-then-else and case are represented using the SELECT - END_SELECT node pairs. The WUR (wait until raised) and WUL (wait until lowered) nodes are used to represent inter-task dependencies. The LEAVE node is used to represent the exit point from loops. Further an operation subgraph can be encompassed within a BEGIN_CONTROL_BLOCK - END_CONTROL_BLOCK node pair. The semantics of these will be become clear in the following section.

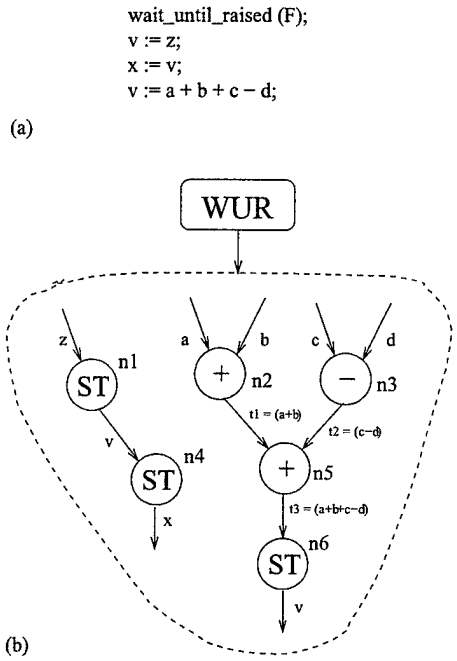


Figure 2.8: **Example Flow Graph**

- *Simple Nodes:* These represent relational, arithmetic, and logical operations. They are binary operations and an expression can be decomposed into a tree of nodes as shown in Figure 2.8. Also, a ST node is used to represent the assignment of a value to a storage element.

The flow graph shown in Figure 2.8 captures only the flow of data, but fails to capture all the concurrencies among the operations. This can be done by performing a detailed dependency analysis.

Dependency analysis The edges in the CDFG represent dependencies among the nodes. Dependencies can broadly be classified as: data and control dependencies. Data dependencies capture the flow of data and hence the order of execution assignment statements. Control dependencies capture the semantics of sequencing, conditional, and loop constructs. Both these dependencies should be properly represented in a correct CDFG representation.

A data_dependency_set is a set of 2-tuples, (n,d) where n is an ancestor node and d is one of the three types of data dependencies (described below) between the two nodes. In the following paragraphs we will discuss data and control dependencies in some detail. [2] provides a more detailed discussion on dependencies in programming languages.

Data Dependencies: Given any two operations represented by nodes N1 and N2, where N1 is a predecessor or ancestor of N2 in the flow graph, we say that:

$$N_2 \; is \; flow \; dependent \; on \; N_1 \; if \;\; output\_set(N_1) \cap input\_set(N_2) \neq \emptyset \qquad (2.1)$$

$$N_2 \textit{ is anti dependent on } N_1 \textit{ if } \textit{input\_set}(N_1) \cap \textit{output\_set}(N_2) \neq \emptyset \tag{2.2}$$

$$N_2 \textit{ is output dependent on } N_1 \textit{ if } \textit{output\_set}(N_1) \cap \textit{output\_set}(N_2) \neq \emptyset \tag{2.3}$$

A data dependency is said to exist between two nodes in the operation graph if any of the above three types of dependencies holds between them. These dependencies can be extracted from VHDL variable assignment statements. Consider a VHDL specification where A and B are any two sequential variable assignment statements with A occurring before B. B is said to be flow dependent on A if a variable that is written to in A is read in B. On the other hand, if the variable is first read in A and later written to in B, then B is said to be anti-dependent on A. If the same variable is written to in both A and B, B is said to be output dependent on A.

Control Dependencies: Gajski, Dutt, and Wu in [10] proposed a way to handle control constructs, by mapping a control-flow representation to an equivalent data-flow representation. This has the advantage of making the concurrencies in the design explicit but generates a complicated graph representation that could get too large and has little correspondence to the original specification. In this section, we will describe a representation that efficiently incorporates control constructs and alleviates some of the problems that arise in a pure data flow style representation.

We will handle control dependencies with the aid of the *control block* demarcated by BEGIN_CONTROL_BLK - END_CONTROL_BLK node pairs. A *control block* is a suitably chosen subgraph of an operation flow graph with no other control nodes in it. Thus, edges within a *control block* always denote data flow. A *control block* is said to have executed if the control flow reaches its END_CONTROL_BLK node. Control flow enters a *control block* only after all previous control blocks have executed.

We can thus view the CDFG of any specification as a series of control blocks. A control block automatically enforces a control dependency, and thus reduces the overhead of maintaining input, output, and dependency sets of the nodes in the operation graph. This simplifies, to a large extent, the complexities involved in generating dependencies although there is some loss in exploiting to the fullest the concurrencies present in the design[2].

The dependency graph for the VHDL specification of Figure 2.8 after dependency analysis is shown in Figure 2.9. Observe the differences between this representation and the earlier flow graph representation. The subgraphs enclosed by dashed lines show the control blocks. The WUR node specifies a control boundary. Therefore the control block encapsulating the first subgraph ends before the WUR node and the second subgraph falls within a new control block. The control block ensures that all preceding nodes are executed before the WUR node is executed and all succeeding nodes are executed only after the WUR node is executed. The operation graph correctly detects dependencies between nodes otherwise missed by the flow graph representation.

From the flow graph in Figure 2.8, it appears that nodes *n4* and *n6* can be executed in parallel but the anti-dependence between them is clearly captured in the dependency graph in Figure 2.9. Further, output dependency between nodes *n1* and *n6* now enforces an order in their execution, even in the absence of node *n4*. On the other hand, no order is specified in the execution of the

[2]This loss can be recovered to a large extent by the use of suitable behavioral transformations that facilitate code motion across control blocks [18]

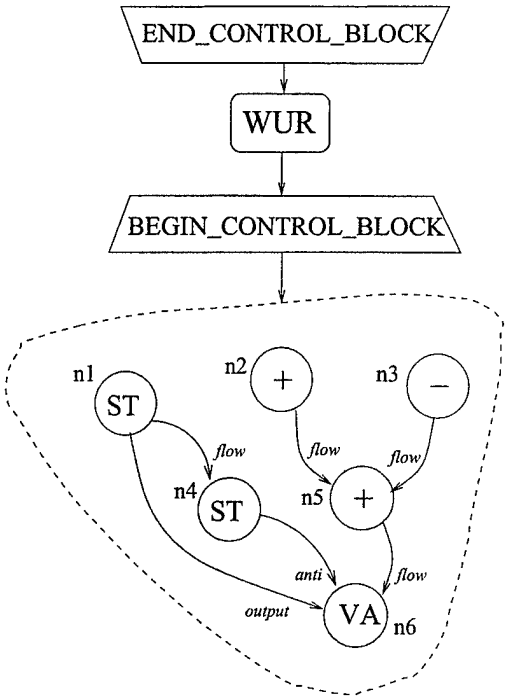


Figure 2.9: **Dependency Graph**

nodes in the set $\{n1, n2, n3\}$ and in the set $\{n4, n5\}$. As a result, the nodes in each of these sets can be executed in parallel. Thus, dependency analysis serves to faithfully capture the semantics and the concurrencies in the specification.

In the following sections we will explain in some detail the translation of conditionals, loops, and wait constructs into a CDFG.

Translation of Conditionals, Loops, and Wait Constructs *Conditionals:* The most popularly used conditional constructs are the 'if–then–else' statement and the 'case' statement. These two are easily translated with the help of control blocks, the semantics of which was explained earlier in Section 2.3.

Case Statement: A typical case statement is shown in Figure 2.10(a). The translated operation graph of the case statement is shown in Figure 2.10(c). The case statement is translated into an expression tree followed by a SELECT node whose input set has the final data flow edge in the expression tree. Therefore, the select node is flow dependent on the final node of the case expression tree. The select node has several branches with a set of select values for each branch. The control flows only into that branch whose set of values match with the result of the case expression. The select node in an operation graph represents this comparison operation that determines the choice of the branch for control flow.

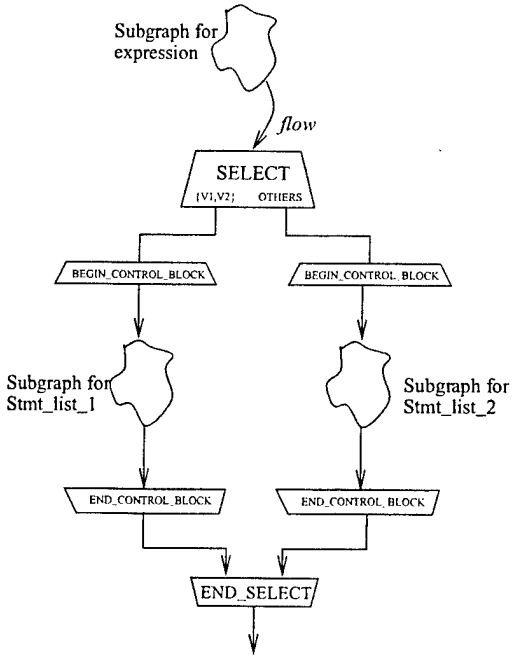
The statements in a case branch are translated into a subgraph, which is enclosed in a control block, as shown in Figure 2.10.

```
case <expression>
    when v1 | v2 =>
        <stmt_list_1>
    when others =>
        <stmt_list_2>
end case ;
```

(a)

```
if <conditional_expression>
then
    <branch-1>
else
    <branch-2>
end if;
```

(b)



(c)

Figure 2.10: **Conditional Constructs**

Therefore, the select node corresponding to a case statement would have as many branches as are in the case statement, each branch having a control block. This ensures that the case expression and the branching operation (represented by the select node) are executed before control can flow into any one of the branches. This enforced control dependency also helps in restricting data dependency analysis within each branch.

If–then–else Statement: A typical 'if–then–else' statement is shown in Figure 2.10(b). An 'if–then–else' statement can be treated as a case statement with two branches and the case expression replaced by the conditional-expression. The select values for the first and the second branches are the Boolean constants TRUE and FALSE, respectively. Any other kind of branching constructs like the 'if–then–elsif–elsif–else–end if' construct can be transformed into an 'if–then–else' statement, the else part of which having another 'if–then–else' statement.

Loops: Loops are usually of three types. The infinite loop with no loop condition, the 'for' loop, and the 'while' loop. All three loop variations can be represented by the generic form shown in Figure 2.11.

We will now describe the translation for the generic loop statement whose operation graph is shown in Figure 2.11. The loop condition is translated into an expression tree followed by a select node whose input is the resulting edge of the tree. The select node has two branches: (1) The true select branch has the subgraph for the loop_body enclosed in a control block. (2) The false select branch has a LEAVE node within a control block. These control blocks ensure that the loop condition evaluation takes place before the control flows into any of these branches. Semantics of the loop-block imply that control flow keeps looping inside until the loop condition

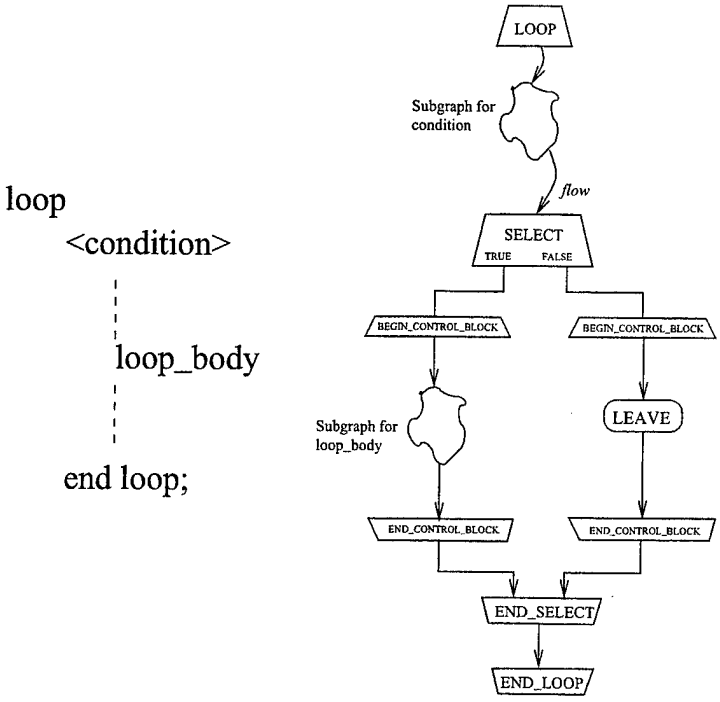


Figure 2.11: **Loop Statement**

evaluates to false and the leave node is reached.

Wait: The Wait_Until_Raised (WUR) and Wait_Until_Lowered (WUL) are implemented as follows:

WUR(flag) = while (not(IsRaised(flag))) nop;

WUL(flag) = while (IsRaised(flag)) nop;

Essentially, both the waits are special cases of the loop construct, where the loop condition subgraph has a simple comparison operation for the raised/lowered value of the flag, and the subgraph for the loop body is empty.

2.3.3 Summary of USM Specification

In this section, we presented an overview of the Uniform Specification Model. The USM provides a hierarchical representation to succinctly capture inter-task level control and dataflow, as well as intra-task operation-level dependencies. More importantly in this work, the USM allows easy representation of the data structures in an application and provides a good memory synthesis environment. Data structures are clearly delimited and defined in the USM; resource arbitration is easily achievable in the USM; and the USM allows memory mapping across different levels of design abstraction. The BBIF along with the USM specification styles provide an adequate environment to clearly characterize data structures of a design and allow efficient mapping and synthesis.

Chapter 3

Temporal Partitioning

3.1 Introduction

Reconfigurable Field Programmable Gate Arrays (FPGAs) [29, 28, 88] built of SRAM-based logic provide designers with *flexible* computing systems. In these devices, the state of the internal static memory cells determines the logic functions and interconnections resident within the FPGA device. This uncommitted array of programmable logic and interconnect on these devices allows reconfiguration between algorithm implementation on the devices. This advantage of FPGAs over Application Specific Integrated Circuits (ASICs) allows the user to use the same circuitry for completely different algorithms by configuring the device between applications. This design approach is generally referred to as *Compile-Time* or *Static Reconfiguration* [89]. Statically configured FPGAs have been used successfully in the rapid prototyping of designs [90, 91]. The long fabrication times associated with ASIC design is eliminated. But the device capacity of FPGAs is far less than that of ASIC chips. Therefore, when synthesizing large designs on FPGAs, usually multi-FPGA boards are used to increase device capacity. This necessitates spatial partitioning of the application. In this style of static FPGA design, the FPGA is configured once at the start of the application, and the same configuration continues till the execution ends.

However, by extending the idea of reconfiguration to intra-application reconfiguration an application which does not fit on the device is divided into multiple segments and multiple configurations of the same application are loaded at run-time. This technique is referred to as *Run-Time or Dynamic Reconfiguration (RTR)* [89]. Current design tools provide support for static reconfiguration, but little tool support exists for dynamic reconfiguration.

When a design is partitioned into mutually exclusive partitions that will execute serially on the reconfigurable processor, the design uses Global Run Time Reconfiguration. All modern FPGAs, whether fully (XC4000, XCV000) or partially (XC6200, XCV000) reprogrammable, can support this reconfiguration step. In partially programmable FPGAs the inactive parts of the FPGA can be reconfigured at run-time even while other parts of the FPGA are active. This flexibility of partial reconfiguration can be exploited in a design approach where subsets of the application are reconfigured as the application executes. This can reduce the time to reconfigure the FPGAs by making it possible to load only the necessary parts of the FPGA. However this increased flexibility also introduces a lot of complexity in the CAD process needed to design applications for such a design style.

The temporal partitioning approach that we have undertaken focuses on generating *global* run time reconfigured designs from behavior specifications of the design. We perform run time reconfiguration in which the entire device is reprogrammed at the boundaries of the temporal segments and data is passed from one temporal segment to the next through a RAM which is not part of the reconfigurable logic. Due to this, structural design is not necessary; behavioral synthesis can be effectively used.

A shortcoming of current automated temporal partitioning techniques is that they choose the underlying implementation of the components of their design before partitioning is performed. Since there are multiple implementations of the components of the designs that vary in the area/delay, it would be more effective to choose the design implementation while partitioning the design by exploring different design options. The search of the design space while partitioning would lead to better partitioned designs.

Due to very high reconfiguration overheads for commercially available reconfigurable hardware, existing automated temporal partitioning techniques [139, 95, 99, 104, 107] usually focus on reducing the latency of the temporally partitioned design by minimizing the number of temporal partitions in the design. But, many DSP applications process an infinite or semi-infinite stream of input data. We will demonstrate that the design with minimum latency *may not* be the best overall solution if we can process multiple inputs on each temporal partition. This technique, called *block-processing* can be used to reduce the the reconfiguration overhead.

If the reconfiguration overhead is ignored, the latency of a temporally partitioned design is usually less than the latency of a static design due to the larger area available. But since the reconfiguration overhead is an important factor in determining the run time of a design, an RTR system may perform poorly as compared to a static design if the reconfiguration overhead dominates the execution time of the design. To overcome the effects of high reconfiguration overhead, we demonstrate [105] how block processing can be introduced at a post-processing step after temporally partitioning a design to increase the throughput. In the current work, we develop a temporal partitioning technique to incorporate block-processing and design space exploration and demonstrate how this integrated processing can be used to search for optimal temporally partitioned designs. In this chapter, an Integer Linear Programming (ILP) based integrated temporal partitioning and design space exploration technique forms a core solution method. For small sized design problems we solve the ILP model to obtain an optimal solution, and we demonstrate the effectiveness of our technique with experimental results. To handle large design problems with our technique we also present an iterative refinement procedure that iteratively explores different regions of the design space and leads to reduction in the execution time of the partitioned design. The ILP based integrated temporal partitioning and design space exploration technique forms a core solution method which is used in a constraint satisfying approach to explore different regions of the design space. Again, we demonstrate the effectiveness of this technique with experimental results.

We present the motivation of our work in Section 3.2, previous work in Section 3.3, the design flow of our tool in Section 3.4, the architecture model, design process model and memory model in Section 3.5, the ILP model, the optimal search algorithm and its results in Section 3.6, the iterative algorithm and its experimental results in Section 3.7, results on random graphs and comparison with other works in Section 3.8, some discussions on extensions and limitations in Section 3.9, and the conclusions in Section 3.10.

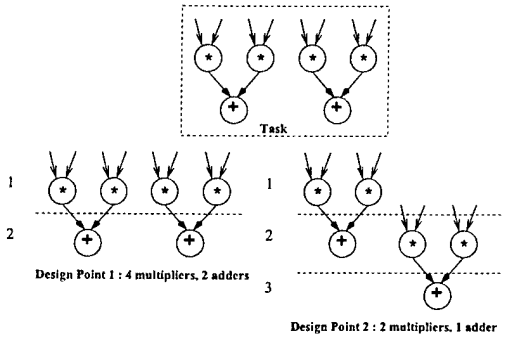


Figure 3.1: **Multiple design points for a task**

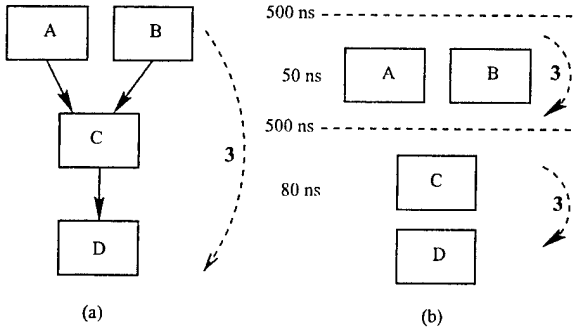


Figure 3.2: **Temporally partitioned design example**

3.2 Motivation

In the following discussion we present the problem of task level design space exploration in temporal partitioning and how its integration with block-processing techniques can improve the execution time of an RTR design.

Input Specification as Task Graphs: Growing design complexity has led designers to generate designs at higher levels of abstraction, such as the behavior level. The designer can concentrate on the required behavior of the application, rather than its implementation. Also simulation at behavior level is much faster than Register Transfer level (RTL) or gate level simulation. In this chapter, we concentrate on behavior level design descriptions to be temporally partitioned. We assume the input specification to be a task graph, where each task consists of a set of operations. Task boundaries can be given by the designer or, tasks can be automatically derived from the behavior specification by clustering or template extraction techniques [110]. Our approach can handle tasks of any level of granularity.

Design Alternatives for Tasks: Depending on the resource/area constraint for the design, different implementations of the same task which represent different area-time tradeoff points can be contemplated. These different implementations are *design points/Pareto points* [113] in the design space of a task. In Figure 3.1, a task and two different implementations of the task are represented. Design Point 1 uses two adders and four multipliers, and is scheduled in two control steps. Design point 2, on the other hand uses less resources and more control steps. If a task is implemented with less resources then the operations in the task will be executed serially, thus increasing the latency of the task. On the other hand, an implementation with more resources reduces the latency but increases the area. Choosing the best design point for each task may not necessarily result in the best overall design for the specification. The most optimal point for a task in the context of optimizing the overall throughput of the design will depend on the architectural constraints of the reconfigurable hardware and the dependency constraints among the tasks. In the subsequent discussion we will express the latency of a design point in terms of total execution time and not in number of clock cycles.

If the number of design alternatives for a task are too many, then exploring the large design space can become too computationally expensive. In such cases, a few 'candidate' design points must be obtained by effective design space pruning techniques, such as discussed in [110]. Since there is a gap between the behavior description and the final synthesized design, it is important that we have accurate synthesis estimates for the tasks. As the size of a task in the task graph is quite small, we use sophisticated High-Level Synthesis estimators which incorporate layout estimation

techniques. Such partitioned designs, can then be predictably taken down to the actual FPGA layout [100, 44].

Block-Processing in Temporally Partitioned Designs: In many application domains eg., Digital Signal Processing, computations are defined on very long streams of input data. In such applications an approach known as *block-processing* is used to increase the throughput of a system through the use of parallelism and pipelining in the area of parallel compilers [114] and VLSI processors [115]. Block-processing is not only beneficial in parallelizing/pipelining of applications, but in all cases where the net cost of processing k samples of data individually is higher than the net cost of processing k samples simultaneously. We can also apply the concept of block-processing to a single processor reconfigurable system to speedup the processing time.

Figure 3.2 illustrates the use of block-processing to speed up computation in a temporally partitioned design. The task graph consists of 4 tasks A, B, C, D. It is partitioned into two temporal partitions as shown in Figure 3.2(b). The latency of temporal partition 1 is 50 ns and of partition 2 is 80 ns. The reconfiguration time is 500 ns. The latency of the design is 50+500+80+500 = 1130 ns. A single iteration of the task graph executes in 1130 ns. Now three iterations of this temporally partitioned design will take 3x1130 = 3390 ns. However if we perform block-processing by sequencing all 3 computations on each temporal partition, the time taken for the execution is (50+50+50)+500+(80+80+80)+500 = 1390 ns. Thus block-processing amortizes the reconfiguration overhead over the 3 computations. Block-processing is possible only for applications that process a large stream of inputs. We represent such applications by a task graph having an implicit outer loop as shown in Figure 3.2(a). Note that block-processing is possible if there are no dependencies among the computations for different inputs. In compiler terminology this means there should be no loop-carried dependencies due to the implicit outer loop, among different iterations of this loop. In this chapter, we deal with applications for which no dependencies among computations is present. Most DSP applications such as Image processing, Template Matching, Encryption algorithms etc. fall in this category. The examples investigated in the RC community include DCT, FFT, DFT, FIR filter and various image averaging, smoothing and filtering algorithms. Also many matrix based computations eg. LU Decomposition for solving linear equations, polynomial interpolation, extrapolation etc. are acyclic in nature.

Integrating Design-Space Exploration and Block-Processing in Temporal Partitioning: For FPGA based architectural synthesis, the constraints of area of the FPGA in terms of CLBs (Configurable Logic Blocks)/FGs (Function Generators) and memory are to be met by the partitioned design. The design alternatives or solutions will vary in the number of temporal partitions and the latency of the partitioned design. For the *spatial* partitioning problem (partitioning of the design for a fixed number of co-existing FPGAs on a board), increasing the number of partitions has the effect of increasing the overall area for the design, and directly affects the latency of the design. Increasing the area, generally increases the number of operations that can execute in parallel (if no dependency constraints exist) and thus decreases the latency of the design. However, for a temporal partitioning system, increasing the number of partitions increases the area available for the design, but this increase is 'over time' and not 'over space'. This increase in number of partitions may or may not result in the reduction of the latency of the design.

When the reconfiguration overhead is very large compared to the execution time of the task it is clear that minimizing the number of temporal partitions will achieve the *smallest latency* in the overall design. In the resultant solution each task will usually be mapped to the smallest area design point among the set of design points for a task. However, it is *not necessary* that the minimum latency design is the best solution. We illustrate this idea with an example. In the

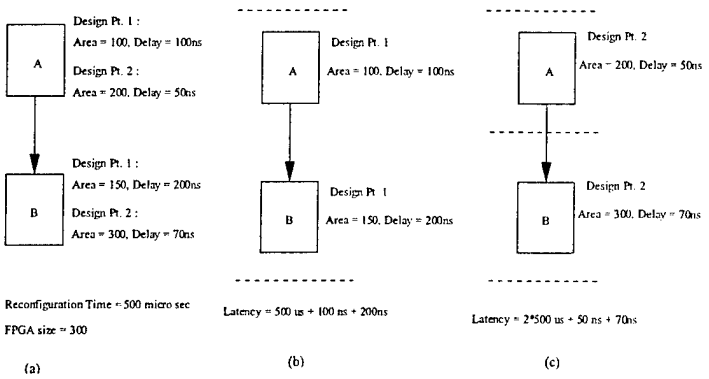


Figure 3.3: **Design space exploration**

Figure 3.3(a) a task graph is shown. Each task has two different design points on which it can be mapped. Two different solutions (b) and (c) are shown. If minimum latency solution is required then solution (b) will be chosen over solution (c) because the latency of (b) is 500.3 μ sec and latency of (c) is 1000.12 μ sec. Now, if we use (b) and (c) in the block-processing framework to process 5000 computations on each temporal partition, then the execution time for solution (b) is 2000 μ seconds and for solution (c) is 1600 μ seconds. Therefore if we can integrate the knowledge about block-processing while design space exploration is being done, then it is possible to choose more appropriate solutions.

The price paid for block-processing is the higher memory requirements for the reconfigured design. We call the number of data samples or inputs to be processed in each temporal partition to be the the *block-processing factor*, k. This is given by the user and is the minimum number of input data computations that this design will execute for typical runs of the application. The amount of block-processing is limited by the amount of memory available to store the intermediate results.

3.3 Previous Work

Design for reconfigurable architectures involves temporal and spatial partitioning and synthesis [100]. There has been significant research on spatial partitioning [101, 102, 103] and synthesis [44, 111], though the research on temporal partitioning is in a nascent stage. Currently many designers perform temporal partitioning manually [92, 93] or the designer needs to specify the partitioning points of the application to the partitioning tool [96]. Luk, Shirazi and Cheung [97] take advantage of the partial reconfiguration capability of FPGAs and automate techniques of identifying and mapping reconfigurable regions from pre-temporally partitioned circuits. Chu et. al [98] present a partial evaluation technique in their circuit generator. In this technique the programmer can provide partial evaluation routines for his components. These partial evaluation routines can then be used to reduce the complexity of the component based on its inputs available at run-time. This technique also utilizes the partial programming capability of the FPGAs, however the programmer has to explicitly define the components that can be partially evaluated and also the method to do so.

Existing automated temporal partitioning techniques, extend scheduling and clustering techniques of high-level synthesis [139, 95, 99, 107] and focus on minimizing the number of partitions of the design. In [95, 99, 107] the temporal partitioning technique involves partitioning gate-level

designs. Since the design to be partitioned is already synthesized, different synthesis options for achieving partitioned solutions with lower execution times cannot be explored. Since the reconfiguration overhead for currently available hardware is very large and dominates the latency of the design, we need to concentrate on techniques to minimize the effects of the reconfiguration overhead. We present an automated technique for DSP style applications, that automatically sequences multiple computations in each temporal partition to reduce the reconfiguration overhead. To our knowledge, no existing tools perform automated block-processing techniques to reduce the reconfiguration overhead in the context of reconfigurable processors. Our technique can also simultaneously handle multiple design constraints, eg., FPGA resources, on-board memories, and perform design exploration that cannot be handled by current techniques in [139, 95, 99, 107]. Kaul and Vemuri [104] presented a mathematical model for combined temporal partitioning and synthesis. In this approach, synthesis cost exploration is performed at an operation level in the task graph, and the number of alternative solutions explored becomes very large. This approach can be used to synthesize small-scale behavior specifications. Kaul and Vemuri also demonstrated the technique of integrated temporal partitioning and design-space exploration for large design problems by using an iterative constraint satisfaction approach [36]. The design space exploration was performed without considering block-processing, so the goal of the system was to minimize the latency of the design.

Wirthlin and Hutchings [94] developed an automated technique that uses partial reconfiguration to load custom instructions at run-time. The instructions in an application are loaded in a demand-driven manner, and unused instructions are removed. This work is in contrast to our approach as it performs the loading and unloading of instructions at run time, whereas in our approach the partitioning into global configurations is performed before the design executes and not at run time.

Kalavade [109] presents an extended bi-partitioning problem for co-design, where partitioning and design point selection is performed sequentially, unlike our combined approach. Interger Linear Programming (ILP) models of other partitioning and synthesis problems have been addressed by researchers. Simultaneous spatial partitioning and synthesis is formulated as an ILP by Gebotys in [106]. Niemann and Marwedel [108] present an ILP-based methodology for hardware software partitioning of co-design systems. Resource constrained scheduling and binding at operation level for ASICs has been formulated as an ILP by Gebotys in [145].

Our temporal partitioning approach is for a globally configured system and we do not consider the partial reconfiguration approach for designing a RTR system. We attempt to perform temporal partitioning at a high-level together with design-space exploration. No other approach to temporal partitioning has attempted to do so. The disadvantage of our technique is that it cannot make use of the partial reconfiguration capability of the FPGAs. This would involve FPGA-specific tools as the different FPGAs have different kinds of partial reconfiguration capabilities. However, our current work is focussed on developing a general purpose tool that can be used to develop temporal partitioned systems for any class of FPGAs on which global reconfiguration can be performed. Some of the partial reconfiguration techniques [97] assume that temporal partitions already exists when they attempt to find matching circuits across temporal partitions. Our technique can be used to automatically generate the temporal partitions that can then be used by such techniques to generate partial reconfigurations.

Contribution of this work: The current work makes several important contributions to the area of reconfigurable design synthesis. It has the following primary features that distinguish it from other works:

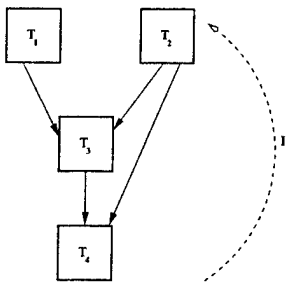


Figure 3.4: **Behavior task graph with implicit outer loop**

- We have integrated the problem of design space exploration into partitioning by using the idea of considering multiple design points for each task in the task-graph. This reduces the complexity of the design space search for the high-level synthesis process by making it concentrate on small portions of the design.
- Our approach performs design space exploration at the behavior level of abstraction, so that multiple design options are explored while performing temporal partitioning and appropriate design points based on the constraints of the architecture are chosen.
- Unlike traditional approaches that concentrate on minimizing the number of temporal partitions of the design, our approach introduces a novel concept of block-processing multiple computations to reduce the reconfiguration overhead and demonstrates that a temporal partitioning approach which combines block-processing and design-space exploration can reduce the design execution time.
- By using ILP as the core engine in an iterative search process we have the flexibility to produce optimal/near-optimal partitioned designs. User controlled parameters influence the search process. If the search for an optimal solution is too time intensive, then suitable search parameters can be given to produce near-optimal results in less run-time.

3.4 System Design Flow

The input specification, is a behavior level design description of the application to be implemented on the reconfigurable hardware. The input specification is shown in Figure 3.4. In Figure 3.5, we present the design flow for building a Run-Time Reconfigured (RTR) design. It consists of an acyclic data flow task graph, with an outer implicit loop. The implicit loop signifies the successive items of input data that will be executed on this task graph. There are no inter-loop dependencies in the task graph due to this implicit loop, i.e., the processing of each input data is independent of any other input. Thus it is possible to perform block-processing for this task graph.

Task Estimation: First, the behavior level estimation engine, which is part of the SPARCS design environment [100], generates multiple design points for each task separately based on the architecture and user constraints. The architecture constraints are the resources available on the reconfigurable hardware, the user constraints are the maximum clock-width for the design. The High-Level Synthesis (HLS) tool makes use of a component library, characterized for the particular reconfigurable processor, to estimate the resource and delay.

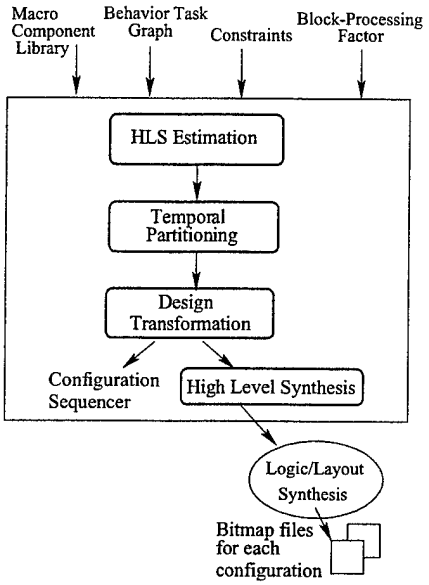


Figure 3.5: **System design flow**

Temporal Partitioning: Next, the temporal partitioning tool divides the task graph into multiple temporal segments, while mapping each task to its appropriate design point. We discuss the ILP formulation used to solve the multi-constraint temporal partitioning problem later in detail.

Design Transformation: Some design transformations are needed so that block-processing can be performed on each temporal partition. This design transformation and the software code to sequence the configurations from the host is generated in this step.

High Level Synthesis: The high level synthesis system in SPARCS [100] is used to generate the RTL design for each temporal segment.

Logic/Layout Synthesis: We use commercial tools, for logic synthesis (Synplify tools from Synplicity) and layout synthesis (Xilinx M1 tools) to convert the RTL description of each configuration into bitmap files.

3.5 Architecture, Design Process, and Memory Model

3.5.1 Reconfigurable Architecture Model

In Figure 3.6, the reconfigurable architecture on which the Run-Time Reconfigured (RTR) design is to be mapped is shown. It consists of a reconfigurable hardware communicating with an external memory. Each temporal partition is mapped to the reconfigurable hardware, and the data flowing between two temporal partitions is mapped to the memory. The host stores all the temporal configurations. It interacts with the reconfigurable hardware to load new configurations and with the memory to load input data and retrieve output data at the end of the execution of the design. Except for the first and last temporal configuration it does not read or write to the memory at any other intermediate configuration.

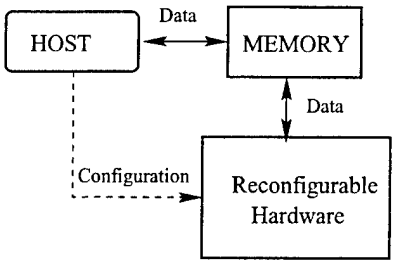


Figure 3.6: **RTR architecture model**

3.5.2 Design Process Model

- Each behavior specification is in the form of a acyclic task graph. A task however has no restrictions and can contain any control structure within it. Each task is indivisible and parts of a task cannot be mapped across partitions.
- Each task has a set of distinct implementation options called design points. These are usually obtained by a high level estimation tool or can be specified by the user. No possible restrictions on the implementation of a task is required, only that the area and delay associated with each design point should be available.
- The high level synthesis process that will be used to synthesize the tasks in each temporal partition is expected to parallelize the intermediate memory transfer with the execution of operations in the task graph. We are assuming that there is enough slack available to do so. Therefore we do not add the intermediate data transfer time to the execution time of the design. If a simple synthesis system is used that does all the memory access in serial with the operations, then we need to add the memory access times in the execution time of the design. We have discussed this further in Section 3.9.
- The estimation process that develops the design point should be closely related to the actual synthesis process that will synthesize the temporal partition after the partitioning is performed. For our design process this implies that the area of the design point should reflect the data path, controller and routing resources required for the task. Xu and Kurdahi [44] and Ouaiss et. al [100] discuss some of the estimation techniques that incorporate low level details in the estimation process. However, if the estimation process is not a true reflection of the ultimate synthesis process then it is required that the user of our system should generate experimentally a factor that reflects the deviation of the actual values from the estimated ones. The area of the FPGA should be reduced by this factor.

3.5.3 Memory Model

- The host writes to the memory of the reconfigurable architecture before the start of the first configuration to place all the data that is to be read as input by the design, and reads from the last configuration's memory all the output data of the design.
- The intermediate data that is needed to be transferred from one configuration to another is written into the memory of the reconfigurable architecture.

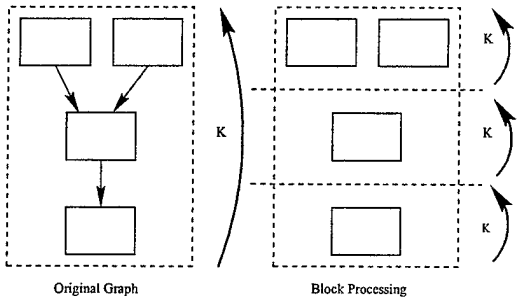


Figure 3.7: **Block-processing model**

- All data to be read in a configuration and written in a configuration is alive for the existence of the whole configuration. The input data from the host is present from the first configuration till the last configuration it is read from. The output data to the host is present from the configuration it is written till the last configuration. Data written in a configuration will remain alive in all subsequent configurations till it is consumed.

- The model for block processing is shown in Figure 3.7. Each configuration processes the whole block of k computations completely and stores the intermediate data. This is repeated for all configurations. Currently, we do not support pipelining of the different computations in the same temporal partition. Therefore the delay for k computations is then equal to k*delay for processing one computation.

3.6 Temporal Partitioning and Design Space Exploration by an Optimal Search Algorithm

The inputs to our Temporal Partitioning system are - (1) Behavior Specifications (2) Target Architecture Parameters (3) Block-processing Factor.
In formal notation, the inputs are stated as -

T	*set of tasks in the task graph.*
$t_i \rightarrow t_j$	*a directed edge between tasks,* $t_i, t_j \in T$, *exists in the task graph.*
$B(t_i, t_j)$	*number of data units to be communicated between tasks* t_i *and* t_j.
$B(env, t_j)$	*number of data units to be read by task* t_j *from the environment.*
$B(t_i, env)$	*number of data units to be written from task* t_i *to the environment.*
R_{max}	*resource capacity of the reconfigurable processor.*
M_{max}	*memory size of the RTR architecture.*
C_T	*reconfiguration time of the reconfigurable processor.*
k	*the block-processing factor for the design.*

The behavior specifications are in the form of a directed graph called the *Task Graph.* The vertices in the graph denote tasks, and the edges denote the dependency among tasks. Data communicated between two tasks, $B(t_i, t_j)$, will have to be stored in the on-board memory of the processor, if the two tasks connected by an edge are placed in different temporal partitions. The target architecture parameters specify the underlying resources and the reconfiguration time, C_T, for the device. Typically, resource capacity, R_{max}, is the combinational logic blocks/function generators on the FPGAs of the reconfigurable device. M_{max}, is the memory for storage of intermediate data available on the reconfigurable processor. k, the block-processing factor is the

lower bound on the number of computations that this design will usually perform. The total intermediate data for k computations of the task graph has to fit in the memory M_{max} of the RTR processor.The user can give k to be the minimum number of iterations of the implicit loop, I, shown in Figure 3.4 for typical runs of the application.

3.6.1 Preprocessing

Design Point Generation: Each task in the task graph is processed by a design space exploration and estimation tool [100] which is part of a high level synthesis system. The high level estimation tool generates a set of *design points* for each task. Each design point is characterized by its area and latency. Each task t will have a set of estimated design points, M_t. We state this formally as -

M_t *set of design points, m, for a task* $t \in T$.
$R(m)$ *area of a design point* $m \in M_t$.
$D(m)$ *latency of a design point* $m \in M_t$.

Partition Bounds Estimation: To find the number of partitions over which the temporal partitioning solution should be explored we calculate two bounds -
1. Lower Bound: For calculating the lower bound on number of partitions N^l_{min}, we sum the *minimum* area design point, m, for each task. This value divided by the FPGA area will be the minimum number of partitions required to obtain a solution.

$$N^l_{min} = \sum_{t \in T} R(m)/R_{max}, \quad \{m \mid \forall m \in M_t, min(R(m))\} \tag{3.1}$$

2. Upper Bound: Ideally, we would like to establish an upper bound on the number of partitions needed to be explored by the partitioner when the maximum area design point for each task is chosen. However, we cannot accurately establish this upper bound on the maximum number of partitions. This is because if a task is too large to fit in some temporal partition, it must go to a later partition. Then all the descendents of this task also cannot occupy the earlier temporal partition even if they can fit in it because the dependency among the tasks will be violated. This will leave some area on temporal partitions unoccupied due to dependency constraints, and the task graph will not fit even though there is enough area left unoccupied on the partitions. We could have established an upper bound on the maximum number of partitions to be equal to the number of tasks in the task graph. However, this is a very pessimistic bound and usually so many partitions need not be explored. We first define, the minimum number of partitions, N^u_{min}, that need to be explored if the *maximum* area design point for each task is mapped by the partitioner, to be -

$$N^u_{min} = \sum_{t \in T} R(m)/R_{max}, \quad \{m \mid \forall m \in M_t, max(R(m))\} \tag{3.2}$$

To determine the upper bound on the number of temporal partitions that need be explored to get an optimal solution, we define a user controlled parameter γ, called the *Partition Relaxation*. γ defines the number of partitions beyond N^u_{min} that must be explored while searching for better solutions. We have introduced parameter, γ, so that a user can direct the partition space search if the user has more knowledge of the solution to the problem. Or, this evaluation of γ can be be done automatically by the tool using heuristic techniques. Using a heuristic, if we map the maximum area design points for each task we arrive at a solution with partition size N''. This

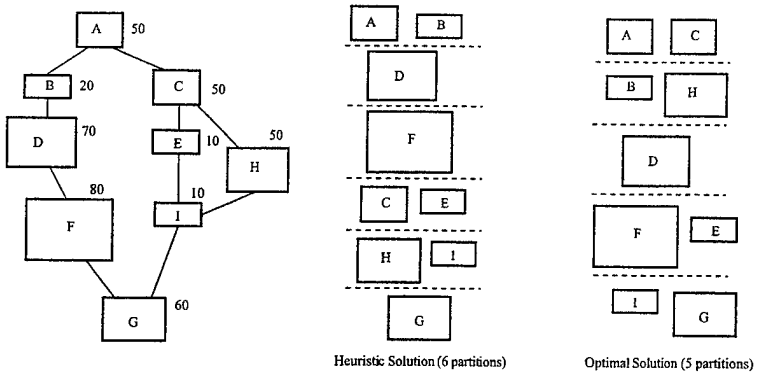


Figure 3.8: **Generation of partition size upper bound**

can be an upper bound on the partition size. If $N'' > N^u_{min}$, then $\gamma = N'' - N^u_{min}$. We give an example of how this can be done in Figure 3.8. A task graph annotated with the maximum area of each task is shown. If R_{max} is 100, then we calculate N^u_{min} to be 4 partitions. A heuristic algorithm maps the tasks as shown into 6 partitions. Therefore $N'' = 6$, and $\gamma = 6 - 4 = 2$. The optimal solution for this graph is obtained in 5 partitions. We claim that any solution obtained by a heuristic using the maximum area (minimum delay) design points will never have its number of partitions less than that of an optimal solution for the same graph. We are currently studying how to achieve tighter upper and lower bounds for partition size and incorporating them automatically in our algorithm. However, the facility of giving γ will still be provided to the user.

In the worst case, the total number of partitions to be explored range from the partition lower bound, N^l_{min}, to the number of tasks in the task graph,$|T|$. Therefore the value of γ can range from 0 to $|T| - N^u_{min}$. We may not get the optimal solution possible for the task graph if the value of γ is not set correctly.

3.6.2 Partition Space Exploration Algorithm

To explore better solutions for the temporal partitioning problem, we need to explore more than one partition bound. The partition bound is the number of partitions for which the current model has been formed and a solution is being explored. Finding the ideal partitions for the overall optimal solution is an iterative procedure, shown in Figure 3.9. Informally, the algorithm consists of the following steps -

1. The starting partition bound is $N = N^l_{min}$.
2. Obtain an optimal solution for the given partition bound, N. The design execution time achieved after solving for this partition bound is D_a. If $N = N^u_{min} + \gamma$, then stop.
3. Increase the partition bound, $N = N + 1$, and reformulate the problem with the new partition size.

Also introduce a design execution time constraint so that the result is bounded by the execution time delay already achieved, D_a. Go to step 2.

We calculate the bounds on the number of partitions, N^l_{min} and N^u_{min}, as described earlier. We start the search at N^l_{min} and obtain an optimal solution, by forming and solving an ILP model of the temporal partitioning problem. The details of the model are described below. For the first ILP model there is no upper bound on the constraint on the execution time of the design. The result of solving this model is a temporal partitioning solution for N partitions and the execution

```
Algorithm Refine_Partition_Bound()
begin
  N^u_min ← MaxAreaPartitions()
  N^l_min ← MinAreaPartitions()
  N ← N^l_min /* starting partition bound */
  FormILPModel() /* Model with no execution time constraint */
  D_a ← SolveILPModel_Optimal()
  while D_a = 0 and N < N^u_min + γ/* Partition bound was infeasible */
    N ← N + 1 /* next partition bound */
    FormILPModel() /* Model with no execution time constraint */
    D_a ← SolveILPModel_Optimal()
  end while
  while N < N^u_min + γ
    N ← N + 1 /* Relax N */
    FormILPModel() /* Model with execution time constraint ≤ D_a */
    D'_a ← SolveILPModel_Optimal()
    if D'_a ≠ 0 /* solution is feasible */
      D_a ← D'_a
    end if
  end while
  return(D_a) /* return with the last known best solution */
end Algorithm Refine_Partition_Bound
```

Figure 3.9: **Partition refinement procedure**

time D_a of the solution. We now relax N by 1, form and solve the ILP model again. This time since we are looking for a better solution than the one we have already achieved, D_a is the execution time constraint for the new ILP model. We continue to relax N and look for better solutions until the value of N reaches $N^u_{min} + \gamma$.

ILP formulation for Design Space Exploration

We build the temporal partitioning model for the given tasks and their design points and the values of N and k. In the following discussion we present the variables and equations of the ILP model.

Variable y_{tpm} models partitioning and design point selection for a task and is described formally as -

$$y_{tpm} = \begin{cases} 1 & \text{if task } t \in T \text{ is placed in partition p, } 1 \leq p \leq \text{N, using design point } m \in M_t \\ 0 & \text{otherwise} \end{cases}$$

where,
N *is bound on the number of partitions.*

Variable y_{tpm} is a 0-1 variable.

Uniqueness Constraint: Each task should be placed in exactly one partition among the N temporal partitions, while selecting one among the various design points for the task.

$$\forall t \in T \quad : \sum_{m \in M_t} \sum_{p=1}^{N} y_{tpm} = 1 \tag{3.3}$$

Temporal Order Constraint: Because we are partitioning over time, a task t_1 on which another task t_2 is dependent cannot be placed in a later partition than the partition in which task

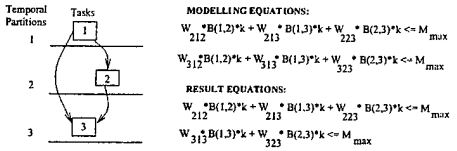


Figure 3.10: **Memory constraint**

t_2 is placed. It has to be placed either in the same partition as t_2 or in an earlier one. This constraint makes sure that the dependency constraints among the tasks are maintained. No task should execute earlier than a task on which it is dependent.

$$\forall t_2,\ \forall t_1 \rightarrow t_2,\ \forall p_2,\ 1 \leq p_2 \leq N-1 \quad : \sum_{m_1 \in M_{t_1}} \sum_{p_2 < p_1 \leq N} y_{t_1 p_1 m_1} + \sum_{m_2 \in M_{t_2}} y_{t_2 p_2 m_2} \leq 1 \qquad (3.4)$$

Resource Constraint: The sum of area costs of all the tasks mapped to a temporal partition must be less than the overall resource constraint of the reconfigurable processor. Typical FPGA resources include function generators, configurable logic blocks etc. Similar equations can be added if multiple resource types exist in the FPGA.

$$\forall p,\ 1 \leq p \leq N \quad : \sum_{m \in M_t} \sum_{t \in T} (y_{tpm} * R(m)) \leq R_{max} \qquad (3.5)$$

Memory Constraint: Intermediate data due to data transfer among dependent tasks will be stored in a partition under two conditions. If the memory has been written in an earlier partition, and is to be read in this partition or any partition later than this partition. Or, if memory is being written in the current partition and is destined to be read in a later partition. Data transfer through memory will not take place if two dependent tasks are placed in the same temporal partition. Variable $w_{pt_1t_2}$ models data transfer requirement across partition boundaries for dependent tasks. It is stated formally as -

$$w_{pt_1t_2} = \begin{cases} 1 & \text{if task } t_1 \text{ is placed in any partition } 1 \cdots p-1 \text{ and } t_2 \text{ is placed in any of} \\ & p \cdots N \text{ and } t_1 \rightarrow t_2 \\ 1 & \text{if task } t_1 \text{ is placed in partition } p \text{ and } t_2 \text{ is placed in any of } p+1 \cdots N \text{ and } t_1 \rightarrow t_2 \\ 0 & \text{otherwise} \end{cases}$$

$w_{pt_1t_2}$ is a 0-1 variable. It is a secondary variable which is described in terms of the y_{tpm} variables.

The intermediate data needs to be stored and should be less than the memory, M_{max}, of the reconfigurable processor. The variable $w_{pt_1t_2}$, if 1, signifies that t_1 and t_2 have a data dependency and are being placed across temporal partition p. Therefore the data being communicated between them, $B(t_1, t_2)$, will have to be stored in the memory of partition p. The following equation represents the memory constraint. It contains terms to represent the intermediate data transfer due to dependent tasks as discussed earlier. Since our memory model is such that all external inputs and outputs with the host also takes place through the memory, the equation also contains terms to represent the amount of data that has to read as input from the environment (host) and written out to the environment (host) by the tasks.

$$\forall p,\ 1 \leq p \leq N \quad : \sum_{t \in T} \sum_{p \leq p_2 \leq N} \sum_{m \in M_t} y_{tp_2m} * B(env, t) * k + \sum_{t \in T} \sum_{1 \leq p_3 \leq p} \sum_{m \in M_t} y_{tp_3m} * B(t, env) * k +$$

$$\sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} (w_{pt_1t_2} * B(t_1, t_2) * k) \leq M_{max} \qquad (3.6)$$

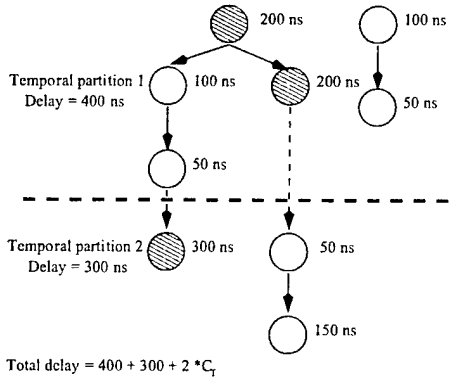


Figure 3.11: **Execution time estimation**

As discussed earlier the variable $w_{pt_1t_2}$ has to model communication among tasks which can be mapped to adjacent and non-adjacent temporal partitions. In Figure 3.10, we show how this variable models data transfer for a small taskgraph fragment. In the example shown there is no data transfer from the host only tasks communicating to each other.We show in the figure the original equations used to model the constraints for Temporal Partitions 2 and 3. The result equations show the $w_{pt_1t_2}$ variables which will be 1 in the mapping of tasks to partitions shown in the example and the constraints which has to be satisfied. $w_{pt_1t_2}$ are non-linear terms and can be generated by the following set of equations -

$$\forall p,\ 1 \leq p \leq N,\ \forall t_2 \in T,\ \forall t_1 \rightarrow t_2,\ : w_{pt_1t_2} \geq \sum_{1 \leq p_1 < p} \sum_{m_1 \in M_{t_1}} y_{t_1p_1m_1} * \sum_{p \leq p_2 \leq N} \sum_{m_2 \in M_{t_2}} y_{t_2p_2m_2} \tag{3.7}$$

$$\forall p,\ 1 \leq p \leq N,\ \forall t_2 \in T,\ \forall t_1 \rightarrow t_2,\ : w_{pt_1t_2} \geq \sum_{m_1 \in M_{t_1}} y_{t_1pm_1} * \sum_{p+1 \leq p_2 \leq N} \sum_{m_2 \in M_{t_2}} y_{t_2p_2m_2} \tag{3.8}$$

Equations (3.7) and (3.8) are non-linear. We can use linearization techniques [121, 122] to transform the non-linear equations into linear ones, so that the model can be solved by a Linear Program solver. Linearization techniques have been used successfully before in [104] to solve the combined temporal partitioning and synthesis problem.

Execution Time Constraint: When the problem is formulated we have as input the partition bound N over which the current solution is to be explored. Variable η is the actual number of partitions finally used in the solution and will be less than or equal to N. Variable d_p models the execution time of a temporal partition.

$$\eta = \text{ Number of partitions actually used in solution.}$$

$$d_p = \text{ execution time of partition } p.$$

η is an integer variable and d_p can be an integer or real variable depending on whether the latency values are integer or real. The following definitions will be used to generate the execution time constraint -

D_a *constraint on the execution time of the design.*

T_l *set of tasks* $t_i \in T$, *where* $\forall t_j \in T, \neg(t_i \rightarrow t_j)$, *(leaf tasks of* T*).*

T_r *set of tasks* $t_j \in T$, *where* $\forall t_i \in T, \neg(t_i \rightarrow t_j)$, *(root tasks of* T*).*

$t_i \xrightarrow{p} t_j$ *a directed path from* $t_i \in T$ *to* $t_j \in T$.

$P_{l \xrightarrow{p} r}$ $\{\ t_i \xrightarrow{p} t_j \mid (t_i \in T_r) \wedge (t_j \in T_l)\}$, *(set of paths from root tasks to leaf tasks).*

The execution time of a partition will be the maximum execution time among all the paths of the task graph mapped to that partition. In Figure 3.11, we show how the execution time for a partition is determined. The final mapping of tasks to partitions, with the latency value for each task, is shown. In partition 1, three paths are mapped. The latency of this partition is the greatest latency along a path mapped to the partition, i.e., maximum among 350ns, 400ns, 150ns. The maximum latency in partition 2 is 300ns. If the block-processing factor is k, then the execution time of the partition is the latency multiplied by the block-processing factor. Formally the execution time of a temporal partition is given as -

$$\forall p,\ \ 1 \leq p \leq N,\ \ \forall (t_i \xrightarrow{p} t_j) \in P_{l \xrightarrow{p} r} \quad : \sum_{m \in M_t} \sum_{t \in t_i \xrightarrow{p} t_j} (y_{tpm} * D(m) * k) \leq d_p \tag{3.9}$$

All temporal partitions $1 \cdots N$ used in the formulation, may not be used in the final solution, if the tasks can fit in lesser number of partitions. To calculate the actual number of partitions used in the solution, we determine the highest numbered partition used by any leaf level task in the task graph by the following equation -

$$\forall t \in T_l \quad : \sum_{m \in M_t} \sum_{p=1}^{N} (p * y_{tpm}) \leq \eta \tag{3.10}$$

Now the execution time constraint on the overall design can be stated in terms of equations (3.9) and (3.10) as -

$$\eta * C_T + \sum_{p=1}^{N} d_p \leq D_a \tag{3.11}$$

As discussed earlier, this constraint is used to search for a better solution as different partition bounds are being explored in Algorithm *Refine_Partition_Bound* in Figure 3.9.

Optimality Goal: The most optimal solution will be the design with the least execution time.

$$Minimize \quad : \eta * C_T + \sum_{p=1}^{N} d_p \tag{3.12}$$

The solution of this ILP model gives us the optimum temporal partitioning for the give partition bound N, the block-processing factor k, and the set of design points for the tasks. If the amount of intermediate memory required to process k computations exceeds the memory constraint M_{max} of the architecture then the user needs to reduce k and temporally partition the design again.

3.6.3 Experimental Results for Optimal Search Algorithm

We performed temporal partitioning on the 4x4 Discrete Cosine Transform (DCT) which is the most computationally intensive part of the JPEG [120] algorithm. In this study, the DCT is a collection of 16 tasks as shown in the Figure 3.12. On the left of the figure we show the internal structure of a task in the DCT. There are two kinds of tasks in the task graph, $T1$ and $T2$, whose structure is similar but whose operations have different bit widths. Task T1 represents two vector multiplications in the first dimension of the DCT. Task T2 represents two vector multiplications in the second dimension of the DCT.We obtained all the design points for each kind of tasks by using estimation tools integrated in the SPARCS design environment [100], on the individual tasks. The functional units, area and latency costs for each is shown in Table 3.1.

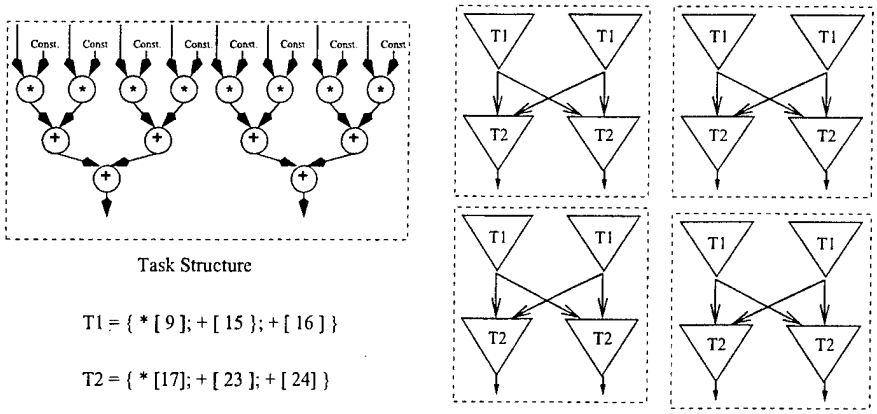


Figure 3.12: **Task graph for DCT**

Table 3.1: **Design points for DCT tasks**

Task	Desgn. Pt.	Characteristics					
		Area (CLBs)	Latency (ns)	$*_9$	$+_{16}$	$*_{16}$	$+_{24}$
T1	1	336	375	8	4	-	-
	2	286	500	6	2	-	-
	3	220	625	4	2	-	-
	4	194	750	2	2	-	-
	5	174	875	1	2	-	-
T2	1	396	420	-	-	8	4
	2	356	560	-	-	6	2
	3	292	700	-	-	4	2
	4	276	840	-	-	2	2

Table 3.2: **Results for combined design-space exploration and block-processing**

Exp.	R_{max}(CLBs)	$C_T(\mu s)$	k	N	Latency (ns)	Design Execution Time	Design Execution Time/k	Mem. Overhead	T(s)
1	4,000	30	3,000	1	31,590	4,800 μs	1,600 ns	0	1
				2	60,795	2,445 μs	815 ns	48,000	1
				3	-	-	-	-	Infeasible
2	4,000	30	1	1	31,590	31,590 ns	31,590 ns	0	1
3	2,304	30	3,000	2	61,590	4,830 μs	1,610 ns	48,000	11
				3	91,215	3,735 μs	1,245 ns	48,000	22
				4	-	-	-	-	Infeasible
4	2,304	30	1	2	61,590	61,590 ns	61,590 ns	16	57

In Tables 3.2 - 3.4 we present the results of our temporal partitioning tool. In all the tables, R_{max} is the resource constraint of the FPGA, C_T the reconfiguration time, k the block-processing factor, and N the number of partitions onto which the design is partitioned. The latency of the final design (with the reconfiguration overhead), is shown in the column Latency. The execution time of the design for the k blocks of data is given in the column Design Execution Time. Design Execution Time/k shows the average execution time per computation, Mem. Overhead shows the amount of maximum memory stored in any of the temporal partitions (excluding the memory used to store the input and outputs) of the solution in terms of the number of words of the hardware. T(s) is the time taken by our temporal partitioning tool to execute in seconds. All experiments were run using an ILP solver called CPLEX on an UltraSparc Machine running at 175 MHz with 120 MB memory.

In Table 3.2, we present the result of temporal partitioning and design space exploration of the DCT with and without block-processing factors. In all experiments the reconfiguration time considered is similar to the Xilinx 6200 series FPGAs. In Exp. 1, for a block-processing factor of 3,000, our temporal partitioning tool explores 3 temporal partitions for the design and results in a latency of 60,795 ns. In Exp. 2, with a block-processing factor of 1 (i.e., no computations are being sequenced), the tool gives a minimum latency design of 31,590 ns and uses just one temporal partition. This results in a statically configured design. Even though, the latency of the statically configured design in Exp. 2 is less than that of Exp. 1, this is not the best possible solution. This is because, if multiple computations are computed on both the static and RTR design, the RTR design will outperform the static design. For executing 3,000 computations, the RTR design will take 2,445 μ sec, while the static design will take 4,800 μsec. This is a 49% improvement of the RTR design over the static design. Exp. 3 and 4 were performed for different FPGA size of 2,304 CLBs, which is the size of a Xilinx XC4062. In Exp. 3, again with a block-processing factor of 3000, the optimal design takes 3 temporal partitions with the latency of the design being 91,215 ns. For Exp. 4, with no block-processing factor the optimal latency of the design is 61,590 ns. Again, the actual execution time of the design when the block-processing factor is considered while exploring the design space is superior. In all the experiments the value of M_{max} is 64 K.

The experiments in Table 3.2 illustrate that combining block-processing and design space exploration gives better temporal partitioning solutions. If the block-processing factor is not considered at the time of temporal partitioning (i.e., is equal to 1), then the temporal partitioning tool will tend to pick the design with minimum number of temporal partitions. If a relevant

Table 3.3: **Results for different reconfiguration overheads**

Exp.	R_{max} (CLBs)	C_T	k	N	Latency (ns)	Design Execution Time	Mem. Overhead	T(s)
5	2,304	30 ns	300	2	1,650	477.06 μs	4,800	17
				3	1,305	364.59 μs	4,800	19
6	2,304	30 ns	50	2	1,650	79.56 μs	700	25
				3	1,305	60.84 μs	700	7
7	2,304	3 ms	3,000	2	6,001,590	10,770 μs	48,000	80
8	2,304	3 ms	30,000	2	6,001,590	53,700 μs	480,000	29
				3	9,001,340	45,450 μs	480,000	36

Table 3.4: **Results for design-space exploration**

Exp.	R_{max} (CLBs)	C_T (μs)	k	N	Latency (ns)	Design Execution Time	Mem. Overhead	T(s)
9	2,304	30	3,000	2	61,715	5,145.6 μs	48,000	1
10	2,304	30	3,000	2	61,590	4,080 μs	48,000	22
				3	91,215	3,735 μs	48,000	204

block-processing factor is given the tool will search for a faster design with more temporal partitions, because block-processing will amortize the effects of reconfiguration overhead. Since we understand that the block-processing is necessary for good performance of a temporally partitioned design, we must integrate this idea early in the design process, while partitioning and design point selection is being performed.

Similar results will hold if the reconfiguration overheads are varied. In Table 3.3, we show results for different reconfiguration overheads. In Exp. 5 and 6, the reconfiguration overhead is in nano-seconds (similar to the reconfiguration overheads of context-switching FPGAs like the Time Multiplexed FPGA [116, 117]). In Exp. 7 and 8, the reconfiguration overhead is in milli-seconds (similar to commercially available reconfigurable hardware, the Wildforce board with Xilinx FPGAs [118]). As the reconfiguration overhead decreases we observe that for small values of k, the exploration process chooses more temporal partitions. However, for the reconfiguration overheads in milli-seconds even for values of k as large as 3,000 the temporal partitioner chooses designs with minimum temporal partitions. So for an architecture which has a very high reconfiguration a large number of blocks must be processed to amortize the cost of the reconfiguration overhead. Such is the case in Exp. 8 where for an overhead of 3 ms, 30,000 computations need to be sequenced to overcome the effect of the reconfiguration overhead and for the tool to partition the design over 3 temporal partitions. From these experiments we see that given the block-processing factor and the architecture constraints the temporal partitioning tool will select the most appropriate design point and the placement of tasks on partitions.

In Table 3.4, we illustrate how design space exploration is beneficial. For same values of the block-processing factor k, we perform experiment with and without design space exploration. In Exp. 9, temporal partitioning is performed with only one design point for each task, the minimal area design point. In Exp. 10, all the design points are used. Again we observe that the tool chooses the most appropriate design points for the given constraints, when multiple design points

are given to it, and results in a 27% improvement of the design in Exp. 10. Therefore design space exploration must be *integrated* with and performed during the temporal partitioning process, rather than choosing the design point *before* temporal partitioning is performed.

The optimal solution process described in this section produced results in less run-times of the temporal partitioning tool when the size of the problem to be solved is not very large. For eg., a task graph of size 15 tasks, 3 temporal partitions and 3 design points per task solved quickly. But a task graph of 30 tasks, 6 temporal partitions and 3 design points per task took many hours to solve.

3.7 Temporal Partitioning and Design Space Exploration by Iterative Search Algorithm

To handle larger problem sizes, we have therefore developed a novel method of solving the ILP problem iteratively. With this method we break the large solution space and window in to smaller regions of the solution space progressively, to obtain near-optimal solutions for the problems. Instead of solving each ILP problem to global optimality we break the search space of the algorithm into smaller sections. An ILP problem for a section of the search space is formed and a constraint satisfying solution is generated. Success or failure of a search guides the algorithm to move iteratively into the next region of search while improving the solution. There can be many ways of dividing the search space into smaller sections. We have approached the problem by dividing the search space by a binary subdivision method.

3.7.1 Preprocessing

In this section, we discuss the additional preprocessing steps which need to be undertaken for the new algorithm that iteratively explores different regions of the design space. The other preprocessing steps of *Design Point Generation* and *Partition bounds Estimation* are as discussed in Section 3.6.

Execution Time Bounds Calculation: The execution time of the temporally partitioned design will involve two components - (1) execution time due to the actual execution of the tasks in each temporal partition for the given block-processing factor k, (2) execution time due to the reconfiguration overhead. For a given number of temporal partitions, N, we can calculate the upper and lower bounds on the execution time of the design as follows -

1. Maximum Execution Time: The worst case execution time D_{max}, will occur when all tasks are serially executed. For upper bound calculation, we will use the design point with maximum execution time for each task. The execution time for each task multiplied to the block-processing factor will give us the execution time of the design without considering the overhead of reconfiguration. This time added to the reconfiguration overhead will be the upper bound design execution time for N partitions.

$$D_{max} = \sum_{t \in T} D(m) * k + N * C_T \tag{3.13}$$

2. Minimum Execution Time: For obtaining the lower bound for N partitions, we consider for each task the fastest (minimum latency) design point. We obtain the latency for all the paths in

```
Algorithm Reduce_ExecutionTime(N, D_max, D_min)
begin
  D_a ← 0
  FormILPModel()
  if SolveILPModel_Feasible() = Infeasible subject to Timeout
    return(D_a)
  D_a ← CalculateSolnDelay() /* Achieved execution time of solution */
  while (D_max − D_min ≥ δ) and (D_a − D_min ≥ δ)
    D'_max = D_max
    /* Binary subdivision of achievable design execution time range */
    D_max = (D_max + D_min)/2
    while (D_max ≥ D_a)
    /* we have already achieved execution time D_a which is less than D_max */
      D_max = (D_max + D_min)/2
    end while
    FormILPModel()
    if SolveILPModel_Feasible() = Infeasible subject to Timeout
      /* increase lower bound to overcome infeasibility */
      D_min = D_max
      D_max = D'_max
    else
      D_a ← CalculateSolnDelay()
    end if
  end while
  return(D_a)
end Algorithm Reduce_ExecutionTime
```

Figure 3.13: **Iterative procedure for reducing design execution time**

the task graph, by summing up the minimum latency of the tasks along each path. The maximum latency value over all such path latencies in the task graph gives us the lower bound on the latency. This latency value is multiplied by the block-processing factor to derive the execution time lower bound without reconfiguration overhead. This execution time added to the reconfiguration overhead will be the lower bound on design execution time for N partitions. In the following equation, p is a path in the task graph.

$$D_{min} = \max_{p} \{ \text{ latency of } p \text{ with fastest design point for each task in } p \} * k + N * C_T \quad (3.14)$$

3.7.2 Algorithm for Design Execution Time Reduction

Figure 3.13, describes the design execution time reduction algorithm. It is an iterative procedure that obtains near-optimal execution time solutions for a given partition bound, N, and execution time bounds D_{max} and D_{min}. The procedure for obtaining appropriate partition bounds was explained in Section 3.6.1. It finds a constraint satisfying solution between D_{max} and D_{min}. Once a solution is obtained, the upper bound is reduced to $(D_{max} + D_{min})/2$, and a new solution for these constraints is found. If a feasible solution is obtained, then the obtained execution time of the solution becomes the upper bound for a new search. If no feasible solution is obtained, then this execution time becomes the new lower bound. It continues this binary subdivision on the execution time bounds, till the difference between the upper and lower bounds becomes very small, or no more feasible solutions are found. The tolerable difference between the lower and

upper execution time bounds for the design is a user defined parameter, δ, called the *Design Execution Time Tolerance*. Design Execution Time Tolerance defines how much of the design space can be left unexplored in one run of the algorithm. If the tolerance is small, more iterations will be spent in obtaining a solution, thus increasing the run time. If a large run time is not acceptable then this tolerance can be increased. The optimality of the solution will be affected by the value of δ. If δ is very large then the algorithm may miss some solution which is better than the one found. We have shown in the experiments the effect of changing the value of δ on the search process. In practice, we can set the execution time tolerance to a small percentage of the $MaxExecutionTime$ of the task graph.

We again use the temporal partitioning and design space exploration problem as modeled as an ILP (presented in Section 3.6.2), with some modifications discussed later. We do not use the ILP for finding optimal solutions, but instead use it to obtain a feasible solution for a problem. That is, the optimization goal explained in Section 3.6.2 is removed and some new constraints are added. These constraints will be presented shortly. Our reduction procedure then makes the constraints tighter, reformulates the ILP and solves it for the new problem. For larger designs, therefore we have developed this directed search procedure, which reduces the search space for each run of the ILP solver, while still exploring the whole search space. This claim has been substantiated, by observing that for small designs the solution obtained by this procedure and an ILP solved to optimality is the same, as discussed in Section 3.6.3. In the algorithm, the procedure $FormILPModel()$ forms the ILP model. The procedure $SolveILPModel\_Feasible()$ then solves the model by a linear program solver and returns with the first feasible constraint satisfying solution.

3.7.3 Partition Space Exploration Algorithm

The partition space exploration procedure for the iterative execution time search is shown in Figure 3.14. It is similar to the partition exploration procedure discussed earlier in Section 3.6.2, the only difference being that the iterative search algorithm $Reduce\_ExecutionTime$ is called to explore different temporal partitioning solutions for each partition bound rather than solving the problem to optimality. Informally, the algorithm consists of the following steps -
1. The starting partition bound is $N = N^l_{min}$.
2. Obtain a constraint satisfying solution for partition bound, N, and execution time constraints D_{max} and D_{min} for this partition bound.
3. Find lower execution time solutions by progressively exploring different regions of the search space, by tightening the execution time constraints, for the current partition bound. If $N = N^u_{min} + \gamma$, then stop.
4. Increase the partition bound, $N = N + 1$, and go to step 2.

3.7.4 Modifications to the ILP model

The ILP model discussed in the Section 3.6.2 remains the same, with some small modifications. We have two execution time constraints instead of Equation (3.11) in the model. These are described below -

$$\eta * C_T + \sum_{p=1}^{N} d_p \leq D_{max} \tag{3.15}$$

```
Algorithm Refine_Partition_Bound()
begin
  N^u_min ← MaxAreaPartitions()
  N^l_min ← MinAreaPartitions()
  N ← N^l_min /* starting partition number */
  D_max ← MaxExecutionTime(N)
  D_min ← MinExecutionTime(N)
  D_a ← Reduce_ExecutionTime(N, D_max, D_min)
  while D_a = 0 /* Partition bound was infeasible */
    N ← N + 1 /* next partition number */
    D_max ← MaxExecutionTime(N)
    D_min ← MinExecutionTime(N)
    D_a ← Reduce_ExecutionTime(N, D_max, D_min)
  end while
  while N < N^u_min + γ
    N ← N + 1 /* Relax N */
    D_min ← MinExecutionTime(N)
    if D_min ≥ D_a
      return(D_a) /* This is the best solution */
      else
      /* find a better solution by taking D_a as upper bound */
      D'_a ← Reduce_ExecutionTime(N, D_a, D_min)
      if D'_a ≠ 0 /* Feasible */
        D_a ← D'_a
      end if
    end if
  while
  return(D_a) /* return with the last known best solution */
end Algorithm Refine_Partition_Bound
```

Figure 3.14: **Partition refinement procedure**

$$\eta * C_T + \sum_{p=1}^{N} d_p \geq D_{min} \tag{3.16}$$

3.7.5 Experimental Results for the Iterative Constraint Satisfaction Algorithm

Case Study of AR filter :

We present a case study of the Auto Regressive (AR) lattice filter [119] that has applications in signal and speech processing applications. In this experiment we demonstrate the closeness of the solution obtained by the iterative constraint satisfaction algorithm presented in this section and the optimal algorithm described in Section 3.6. The task graph for the specification consists of 6 tasks as shown in Figure 3.15. Tasks A and B show the internal structures of the filter tasks. Tasks T1, T3, & T4 have a structure like Task A, but differ in the bit-widths of their operations. Tasks T2 and T5 are like Task B, but again differ in their bit-widths. The bit widths of each operation in each task is also shown in the figure. The design points are shown in Table 3.5. These design points were again estimated using an estimation tool integrated in [100]. Task T1 has three design points, tasks T3 & T4 have two design points each, and tasks T2 and T5 have one design point each. The result of the experimentation is shown in Table 3.6. N denotes the number of temporal partitions explored. The columns under Result(Iterative) state the result of running the iterative algorithm. I is the iteration of the algorithm, D_{max} and D_{min} are the design

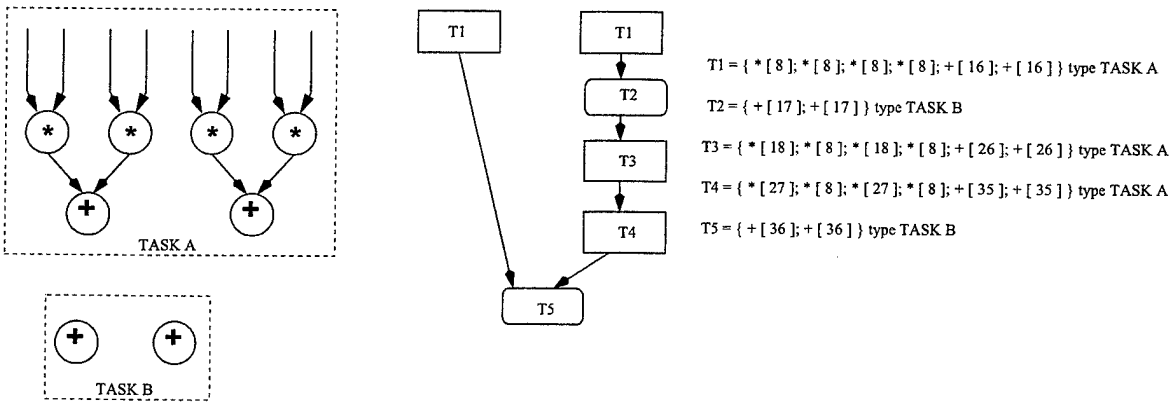


Figure 3.15: **Task graph for the AR filter**

Table 3.5: **Design points for the AR filter tasks**

		Characteristics									
t	M_t	Area	Latency	$*_8$	$+_{16}$	$+_{17}$	$*_{18}$	$+_{26}$	$*_{27}$	$+_{35}$	$+_{36}$
T1	1	120	250	4	2						
	2	104	375	2	2						
	3	84	625	1	1						
T2	1	30	125			2					
T3	1	170	320	2			2	2			
	2	118	480	1			1	1			
T4	1	222	400	2					2	2	
	2	155	600	1					1	1	
T5	1	54	200								2

Table 3.6: **Temporal partitioning of the AR filter,** $R_{max} = 196$, $C_T = 30$ **μs,** $\gamma = 0$, $\delta = 100$ **μs,** $k = 3000$

		Result(Iterative)			Result(Optimal)	
N	I	$D_{max}(\mu s)$	$D_{min}(\mu s)$	Design Execution Time(μs)	Design Execution Time (μs)	Mem. Overhead
3	1	8,055	3,975	Inf.	Inf.	
4	1	8,085	4,005	6,210		
	2	6,045	4,005	5,355		
	3	5,025	4,005	Inf.		
	4	5,280	5,025	Inf.	5,355	15,000
5	1	5,355	4,035	5,010		
	2	4,650	4,035	Inf.		
	3	4,950	4,650	Inf.	5,010	18,000

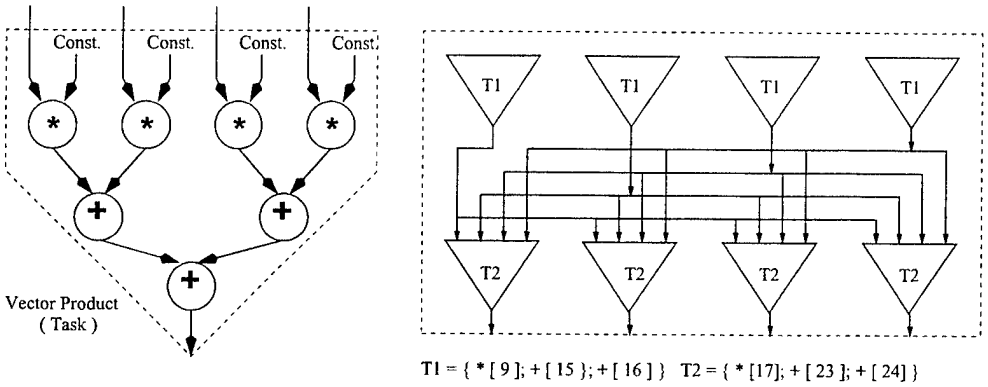


Figure 3.16: **Task graph for DCT, 8 of the 32 tasks are shown**

execution time bounds for that iteration calculated by the algorithm. D_a gives the design execution time of the solution. *Result(Optimal)* is the result achieved by solving the problem to optimality using the algorithm described in Section 3.6. Mem. Overhead shows the amount of maximum memory stored in any of the temporal partitions (excluding the memory used to store the input and outputs) of the solution in terms of the number of words of the hardware. We use *CPLEX* to solve the ILP problems both for constraint satisfaction and optimal solution. We see that the result of our algorithm matches the optimal solution for this task graph. We have performed a lot of experiments on small task graphs and the solution for our iterative procedure and an optimally solved ILP has been the same.

Case Study of DCT : For task graphs with larger number of tasks, the iterative constraint satisfaction approach is able to explore in reasonable time more solution space than solving the problem to optimality. To demonstrate this, we again undertook a case study of the 4x4 DCT, however this time the size of each task is smaller. In this study, DCT was modeled in the form of 32 vector products. The entire DCT is a collection of 32 tasks, where each task is a vector product. A vector product is shown in Figure 3.16. There are two kinds of tasks in the task

Table 3.7: **Design points for DCT tasks**

Task	D.	Characteristics					
		Area	Latency	$*_9$	$+_{16}$	$*_{16}$	$+_{24}$
T1	1	180	375	4	2		
	2	138	500	2	2		
	3	121	750	1	2		
T2	1	216	420			4	2
	2	188	560			2	2
	3	162	840			1	2

graph, $T1$ and $T2$, whose structure is similar to the vector product, but whose bit-widths differ. A collection of eight tasks, forms a row of the 4x4 output matrix, as shown in the figure. The entire task graph consists of four such collections of tasks. Each task had three design points. Area and latency of the tasks for these design points were carefully estimated using an estimation tool [100]. The functional units, area and latency for each is shown in Table 3.7. The result of the iterative refinement procedure for minimizing the design execution time of the temporally partitioned DCT for various FPGA resource bound, R_{max}, and reconfiguration overhead, C_T, values is shown in Tables 3.8 through 3.11. For the current set of experiments, column N denotes the number of temporal partitions, D_{max} and D_{min} denote the maximum and the minimum design execution time bounds for the model being solved in that iteration. The design execution time of the solution produced is shown in the column Design Execution Time. Run times for the temporal partitioning tool, in seconds, are shown for each iteration of the algorithm separately in the column T(s). The total run time of the temporal partitioning tool in minutes for each experiment is shown in column T(m). All experiments have been run on an UltraSparc 1 machine running at 175 Mhz with 120 MB memory.

In the first experiment, shown in Table 3.8, $R_{max} = 576$ CLBS (XC4013 FPGA) and C_T is 30 μs and the block-processing factor $k = 3000$. The minimum number of partitions estimated by $MinAreaPartitions()$ is 8 and by $MaxAreaPartitions()$ is 11. We are able to reduce the execution time of the circuit in steps by doing a binary division. Once the difference between the maximum and minimum execution time is less than $\delta = 1000\mu$s, we stop. Then, the algorithm proceeds by searching the next partition bound by increasing N and repeats the iterative search procedure. We sometimes need to have a timeout, either if the problem is infeasible or a solution is too difficult to find. This timeout is shown in the results as Inf.. For this set of experiments we kept the timeout to be 300 seconds to find each constraint satisfying solution. Notice that, while we are tightening the design execution time constraint in each iteration of the solution, we are in effect making the solver progressively look at different parts of the design space. Since Ending Partition Relaxation, $\gamma = 1$, we stop our search at $N = 12$.

In the second experiment shown in Table 3.9, we present the temporal partitioning of the same design with no block-processing being performed i.e, $k = 1$. For this experiment, we have not shown the value of reconfiguration overhead $N * C_T$ in the table. We start with 8 partitions, but no solution is possible. Then we relax the partition bound by 1, to 9 and continue the search for a solution. Notice that no relaxation of N was undertaken in this experiment, after a solution was achieved in 9 partitions. This is because, the algorithm $Refine\_Partition\_Bound$ calculates the new lower bound, D_{min}, and finds that it is greater than the already achieved execution time, so

Table 3.8: **DCT**, $R_{max} = 576, \delta = 1000\mu s, \gamma = 1, k = 3000$

			Bounds		Result		
$C_T(\mu s)$	N	I	$D_{max}(\mu s)$	$D_{min}(\mu s)$	Design Execution Time(μs)	T(s)	T(m)
30	8	1	76,580	2,625	Inf.	300	
	9	1	76,590	2,655	28,410	37.40	
		2	21,138	2,655	20,640	77.32	
		3	11,895	2,655	Inf.	300	
		4	16,515	11,895	Inf.	300	
		5	18,825	16,515	Inf.	300	
		6	19,980	18,825	Inf.	300	
	10	1	20,640	2,685	18,900	278.8	
		2	11,631	2,685	Inf.	300	
		3	16,104	11,631	Inf.	300	
		4	18,342	16,104	Inf.	300	
		5	18,621	18,342	Inf.	300	
	11	1	18,900	2,715	Inf.	300	
	12	1	18,900	2,745	Inf.	300	61.55

Table 3.9: **DCT**, $R_{max} = 576, \delta = 1000\mu s, \gamma = 1, k = 1$

			Bounds (without $N * C_T$)		Result		
$C_T(\mu s)$	N	I	D_{max}(ns)	D_{min}(ns)	Design Execution Time (without $N * C_T$)(μs)	T(s)	T(m)
30	8	1	25,440	795	Inf.	300	
$\alpha = 0$	9	1	25,440	795	9,630	77.60	
		2	6,956	795	Inf.	300	
		3	9,266	6,956	9,100	78.95	
		4	8,111	6,956	8,100	185.73	
		5	7,533	6,956	7,380	281.93	
		6	7,244	6,956	Inf.	300	25.4

Table 3.10: **DCT**, $R_{max} = 1024, \delta = 1000\mu s, \gamma = 1, k = 3000$

			Bounds		Result		
$C_T(\mu s)$	N	I	$D_{max}(\mu s)$	$D_{min}(\mu s)$	Design Execution Time (μs)	T(s)	T(m)
30	5	1	76,410	2,535	18,240	20.92	
		2	11,775	2,535	Inf.	300	
		3	15,816	11,775	Inf.	300	
		4	17,709	15,816	16,980	288.46	
	6	1	16,980	2,565	11,760	76.43	
		2	9,772	2,565	Inf.	300	
		3	11,574	9,772	Inf.	300	
	7	1	11,760	2,595	11,520	214.4	
		2	7,146	2,595	Inf.	300	
		3	9,483	7,146	Inf.	300	
	8	1	11,520	2,625	Inf.	300	45.00

it stops. Again, if we compare this result with the experiment where we had considered block-processing of designs, we see that the design in Table 3.8 will perform 3000 computations in 18,900 μ seconds while the current design will perform the computations in 22,410 μ seconds. So it is important to integrate both block-processing and design space exploration as part of the temporal partitioning process so that appropriate task mapping to partitions and design points is performed to produce designs that will give better performance

In Table 3.10, we show the results on DCT with $R_{max} = 1024$ (XC4025 FPGA). In this experiment the execution time tolerance δ is 1000 μs. To show how varying the parameter δ affects the performance of the algorithm, we reduce δ to 100 μs and repeat the same experiment whose results are shown in Table 3.11. The number of iterations spent looking for a solution increases, thus increasing the runtime. But a better solution is achieved. We therefore observe that reducing execution time tolerance increases the run time but achieves better solutions. For all the experiments shown in this section, we also experimented with obtaining optimal solutions as we have shown for the AR filter. However, in *none* of these experiments could the optimal solution process get even a single feasible solution in the same run time as the iterative solution process. This is because in the iterative solution process we are dividing the solution space into smaller regions, thus reducing the size of the problem that the ILP solver has to solve in one run of execution. Also, we are directing the search process to look from higher design execution time solutions towards lower design execution time solutions and this directed search process seems to help the solver when solving problems with very large solution spaces.

We have applied this technique to various other examples like 2D-FFT and FIR filter, median filter. The results we noticed are similar and also since the their taskgraphs are very regular like DCT we have instead included results for random unstructured graphs in the next section. This shows the viability of the approach for both regular and non-regular graphs.

3.8 Comparison with List Based Scheduling Algorithm

Following the two case studies we demonstrate the results of our techniques with another temporal partitioning algorithm based on the list scheduling technique. We will compare the results for the

Table 3.11: **DCT**, $R_{max} = 1024, \delta = 100\mu s, \gamma = 1, k = 3000$

			Bounds		Result		
$C_T(\mu s)$	N	I	$D_{max}(\mu s)$	$D_{min}(\mu s)$	Design Execution Time (μs)	T(s)	T(m)
30	5	1	76,410	2,535	18,240	20.92	
		2	11,775	2,535	Inf.	300	
		3	15,816	11,775	Inf.	300	
		4	17,709	15,816	16,980	288.46	
		5	16,761	15,816	16,020	74.17	
		6	15,934	15,816	Inf.	300	
	6	1	16,020	2,565	11,760	76.43	
		2	9,772	2,565	Inf.	300	
		3	11,574	9,772	Inf.	300	
		4	10,359	11,574	10,560	104.04	
		5	10,510	11,574	Inf.	300	
	7	1	10,560	2,595	Inf.	300	
	8	1	10,560	2,625	Inf.	300	49.4

DCT example which is a very regular graph. To find how our algorithm works on unstructured graphs we also generated many random graphs. The characteristic feature in which they differ from DCT is that they are graphs with more tasks in their critical path i.e. they are long graphs. Also the various tasks are different in size and so vary in the number of design points.

We now discuss briefly the the scheduling algorithm which is similar to some other temporal partitioning works in literature [139]. In other partitioning works temporal partitioning is performed on an operation level data flow graph. Each operation in the data flow graph is placed in a priority list honoring the dependency among the operations. The priority list is formed by placing the nodes on the list one by one. A node is placed on the priority list if all its predecessors are already on the priority list. Then the algorithm assigns nodes starting from highest to the lowest priority in a partition until the area is filled. Once a partition is filled nodes are assigned to the next temporal partition. Each operation has one area and delay value associated with it. We will extend the same list based scheduling technique to work on task graphs instead of operation graphs. However there is no easy way to incorporate multiple design points in this technique. Therefore, going by the philosophy of this approach where the aim is to minimize the number of partitions in the design we choose the least area design point for each task prior to the start of the list based scheduling algorithm.

Table 3.12 presents the result of our comparison for the DCT and the random graphs. We have shown the design execution times for our iterative search algorithm and the list based scheduling algorithm. The results are presented for each partition bound for which a solution is generated by the algorithms. Since the reconfiguration overheads for both the algorithms is the same we show the design execution times without the reconfiguration overheads.

Graph Random 1 consists of 20 nodes, Random 2 has 30 nodes. Both the graphs have up to 4 design points per task. We have presented results for different area constraints and block-processing factors. The set of results on the DCT example and the random graphs demonstrate the improvement in performance of our algorithm over the list based scheduling method. The results are for varying block-processing factors. In each of the results the performance of our algorithm is superior by 7%-40%. This demonstrates the following two

Table 3.12: **Comparison with list based scheduling algorithm**

Exp.	R_{max} (CLBs)	C_T (μs)	k	N	Design Execution Time		% improv
					List Based	Iterative	
DCT	1024	30	1	5	7,200ns	4,610ns	35.97
			3,000	5	21,450μs	16,680μs	22.23
			3,000	6	-	11,400μs	46.85
			3,000	7	-	11,110μs	48.20
Random 1	1024	30	1	4	9,000ns	5,100ns	43.3
			3,000	4	27,000μs	14,850μs	45
Random 2	1024	30	1	8	13,500ns	10,950ns	18.88
			3,000	8	40,260μs	27,450μs	31.8
Random 2	2034	30	1	2	6,450ns	4,290ns	33.48
			3,000	2	19,350μs	13,500μs	30.23
				3	-	12,600μs	34.88

significant points -

- Design space exploration without block-processing is meaningful because the exploration process will choose the most appropriate design points for the given constraints. The first line in Table 3.12 with no block processing demonstrates a performance improvement of 35% over the results from an algorithm that chooses the design points prior to the temporal partitioning step.
- Design space exploration with block processing demonstrates that the amortization of the reconfiguration overhead due to block processing will help in the usage of more temporal partitions. For DCT the result demonstrates an up to 48% improvement over the list based algorithm that does not consider block processing.

3.9 Extensions and Limitations of the Work

All our methodology is still applicable in case of inter-loop dependencies by simply setting the block processing factor to '1' (i.e., no block processing). However, in the presence of inter-loop dependencies (i.e., absence of block processing) temporal partitioning is generally not time-effective for the devices like XC4000 that have high reconfiguration time. However, for devices such as XC6200 and the context switching FPGAs, where reconfiguration time is relatively low, temporal partitioning remains viable and useful. In either case our formulation will produce an optimal or near-optimal temporal partitioning solution after performing design space exploration and choosing the most appropriate design point for each task. (This solution may have only one temporal segment for architectures with high reconfiguration overheads.) We now present some of the extensions of our work and the limitations of the current technique.

3.9.1 Intermediate Data Transfer Time

In the design process model in Section 3.5, we have assumed that a suitable high level synthesis system exists that can schedule memory accesses together with the operations in the task graph if there is enough slack available. However if such a synthesis system is not available and the

memory accesses have to be performed prior to the execution of the task graph we need to account for the memory read write access times in our model. In the model presented earlier we have not integrated the calculation of read and write times for intermediate data in the calculation of the delay of each temporal partition. However, our model is very extensible in this respect. Since we are already calculating the amount of data transfer taking place across each temporal partition, the current model can be extended by modifying the minimization goal of the ILP model to include the intermediate memory read-write times. To do this we extend the minimization goal to also include -

Amount of data transfer * (Memory read time + Memory write time).

We can exclude the memory read by the input tasks and memory written to by the output tasks from the minimization goal as this factor is a constant of the graph and cannot be reduced.

In terms of the equations presented in Section 3.6.2, we already have a variable $w_{pt_1t_2}$ defined that is representative of whether data is being transferred across temporal partition p due to tasks t_1 and t_2. To calculate the read and write times for the intermediate data we only need to know, if a data transfer is taking place but are not concerned about the partition boundaries it is taking place. Therefore, we can generate a new variable $i_{t_1t_2}$ that represents the data transfer due to tasks t_1 and t_2 without considering the partition where this transfer takes place. This can be done in terms of the $w_{pt_1t_2}$ variables already generated. Formally, we generate the variable $i_{t_1t_2}$ below -

$$i_{t_1t_2} = \begin{cases} 1 & \text{if task } t_1 \text{ and } t_2 \text{ are not placed in the same temporal partition} \\ 0 & \text{otherwise} \end{cases}$$

$$\forall p,\ 1 \leq p \leq N,\ \forall t_2 \in T,\ \forall t_1 \rightarrow t_2 \quad : i_{t_1t_2} \geq w_{pt_1t_2} \tag{3.17}$$

Then the time required for the data transfer of intermediate data is equal to -

$$i_{t_1t_2} * B(t_1, t_2) * k * D_{mem}$$

where,

D_{mem} *is the sum of the read and write time for one memory element of the reconfigurable processor*

Now the minimization goal for the new problem will be -

$$Minimize \quad : \eta * C_T + \sum_{p=1}^{N} d_p + i_{t_1t_2} * B(t_1, t_2) * k * D_{mem} \tag{3.18}$$

To extend the technique presented in Section 3.7 we need to include the intermediate data read and write time in the generation of the delay bounds D_{max} and D_{min} used in that algorithm. In the preprocessing step where we generate D_{max} and D_{min} we can also generate upper and lower bounds on the amount of data transfer that can take place for the given task graph. The upper bound on the intermediate data transfer is given by the sum of all data transfers that can ever take place in the task graph. This is available by summing all data transfers across all edges in the task graph. This value multiplied by D_{mem} is the upper bound on the time to transfer the intermediate data for the task graph. The upper bound on execution delay of design (as calculated in Section 3.7) + upper bound on the intermediate data transfer time will be the new D_{max} . The lower bound for the data transfer is 0, so D_{min} will remain the same as calculated in Section 3.7.

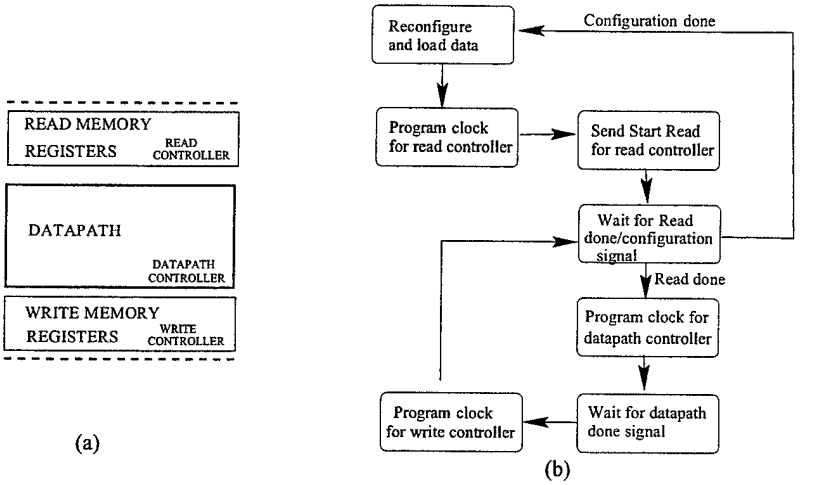


Figure 3.17: **Reduction of time for memory access**

With the above extension, the intermediate data transfer time will be incorporated in the algorithm. The effect on the solution will be twofold. If the memory read/write time for tasks is very small compared to the execution times of the tasks then results similar to our experimental results will still be generated. However, if the memory read/write times are of comparable magnitude, then we will see solutions that have a tendency to avoid cutting across intertask edges. All the intermediate data-transfer time in Equation (3.18) is added to the design execution time. However, in practice, part of this cost can be reduced in the following ways -

- It is not necessary to have a design where all the intermediate data is read and written completely excluded to the execution of the design. Much of this read/write can be performed in parallel to the execution of the rest of the design. We can also develop an estimation process that calculates the overhead of intermediate data transfer for each design point, if such a transfer were to take place because of a task being placed in the next temporal partition. This can be incorporated in our model and accurate execution time results will be generated. This estimation process and model is currently being investigated.

- Or, if the read and write have to be performed in serial to the design execution, we have developed a model that will reduce the time to access memory by generating two separate clocks - one for memory access and one for design execution. Figure 3.17 presents an overview of this approach. If a single clocking scheme is used for the FPGA the clock width is limited by the maximum combinational delay among all the functional units in the design. Usually the memory access can take place at a much faster rate than the clock frequency dictated by the design. Therefore we have split the memory access and the design execution so that memory access can occur at a faster rate. The time to program the clock from the host is usually negligible as it involves writing a single word to the reconfigurable processor.

We can extend Equation (3.18) by multiplying the data-transfer time with a constant 'reduction factor' between 0 and 1 that can be used to appropriately scale down the memory access time by

Table 3.13: **Results for variation of the factor reducing memory access time**

Exp.	R_{max} (CLBs)	Reduction factor	N	Design Execution Time(μs)
DCT D_{mem} = 140 ns	2304	0	2	4,830
			3	3,735
		.2	2	5,408
			3	5,063
		.4	2	6,885
			3	6,423
		.6	2	7,860
AR filter D_{mem} = 140 ns	196	0	4	5,355
			5	5,010
		.2	4	5,940
			5	5,845
		.4	4	6,531

the average amount that it is being reduced by using one of the techniques discussed above. If the factor is 0 then all the memory access costs have been absorbed. If the factor is 1 then none of the costs have been absorbed. We now present a few experimental results in Table 3.13 for different values of this reduction factor for the DCT and AR filter examples. For all the experiments $C_T = 30\mu s$ and $k = 3,000$. We see from the tables that for the DCT temporal partitioning will be explored till the reduction factor is .4. For the reduction factor at .6 the design with minimum number of partitions is the best solution. For AR filter the reduction factor of $\geq .4$ stops the exploration process.

3.9.2 Intermediate Data Overhead

Intuitively we can understand that due to block processing the amount of memory required for saving the intermediate data will be k times the amount of memory required for a temporally partitioned solution that does no block processing. This would happen if a solution generated for $k = 1$ (no block processing) is used to process blocks of data. However, it is not necessary that the solution generated by our algorithm for both block processing and non block processing in a design will give the same results. Formally, we can state the overhead of intermediate memory needed for block processing in each temporal partition in terms of the variables of the ILP model. The total amount of memory overhead in each partition is given by -

$$\sum_{t \in T} \sum_{p \leq p_2 \leq N} \sum_{m \in M_t} y_{tp_2m} * B(env, t) * k + \sum_{t \in T} \sum_{1 \leq p_3 \leq p} \sum_{m \in M_t} y_{tp_3m} * B(t, env) * k + \sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} (w_{pt_1t_2} * B(t_1, t_2) * k) \tag{3.19}$$

The maximum of these values for all partitions would determine the size of the external RAM required for the system.

If we run two versions of a specification through our system, with and without block processing, we can determine the overhead due to block processing. As a byproduct of our model we can thus calculate precisely the memory overhead due to block processing in each design.

3.9.3 Limitations

As we have discussed earlier in Section 3.8, our techniques demonstrate that design space exploration with block processing is beneficial in amortizing the cost of the reconfiguration overhead. However, if data is to be processed in real time where blocks of data are not available a priori, our method can still be used to search for a temporally partitioned solution if one is possible within the inter block-arrival time constraint. It is possible for our system to take the inter block-arrival time constraint on the overall execution time rather than have one generated by the tool. If a static/temporally-partitioned solution is possible it will be generated by our tool. However, if the designer cannot specify an inter block-arrival time or if this time varies for various inputs and cannot be know a priori then our methods cannot be applied.

The current implementation does not support pipelining of the different computations in the same temporal partition. This would be particularly beneficial as it would reduce the execution time for the designs. Another limitation of the approach is that even though tasks can be of arbitrary granularity, splitting of tasks across temporal partitions is not allowed. Currently the memory read/written in a temporal partition remains alive for the life of a temporal partition. More detailed memory access models would require sophisticated foot-print analysis of the memory-bound data structures and is beyond the scope of the current work. The partial RTR capabilities of the reconfigurable device is also not exploited from within the algorithm.

3.10 Conclusion

We presented an automated temporal partitioning methodology, which demonstrates how integrating design space exploration and block-processing procedures, can lead to performance enhancements in dynamically reconfigured designs even when the reconfiguration overhead is a dominating factor in the computation time. We have shown, that by using mathematical programming techniques we can model the task level temporal partitioning and design exploration problem incorporating multiple constraints of area, design execution time, and memory. We have also developed a framework in which these techniques can be used in a novel manner to solve constraint satisfaction problems for large specifications of real world examples such as the DCT. We are able to get near-optimal solutions in short run times with this iterative procedure. The effectiveness of the formulations and iterative procedure was demonstrated by the case study of the DCT.

This technique can handle tasks of arbitrary granularity, so the same technique can be used to handle task graphs with task sizes varying from small to very large. It is also possible to address sharing of resources in a temporal partition though the problem size and complexity will be increased as more variables will be added to the ILP model to model sharing of resources.

Chapter 4

Architecture-Driven Spatial Partitioning

4.1 Introduction

Design process for RCs involves partitioning and synthesis of the given design specification onto the FPGAs and memories on the RC and accordingly establishing the required pin-assignment and inter-FPGA routing. Partitioning of a design may be performed at various levels: behavioral, RTL, or gate-level. High level synthesis (HLS) process converts a behavioral specification into an RTL design having data path (structural net-list of components) and a controller. Behavioral partitioning is a pre-synthesis partitioning while structural partitioning is done after HLS. Studies [38, 48, 20, 19] comparing Behavioral and RTL partitioning show the superiority of the former for large designs.

Gate-level and RTL partitioning are both structural level partitioning problems that are typically modeled as graph partitioning. In RTL partitioning the nodes are components from an RTL library, while in gate-level partitioning the components are from the target specific device library. In fact, the same structural partitioning engine has been used to perform both RTL and gate-level partitioning [42]. Problem sizes for gate level partitioning are a magnitude larger than for RTL partitioning. If the RTL components are pre-placed macros [44, 26] that must not be flattened into gates, then gate level partitioning is not performed. Usually gate-level partitioning is used in the context of certain placement algorithms that use recursive partitioning strategies to minimize the wire length [47].

Behavioral partitioners must be guided by high-level estimators that make estimates on device area, memory size, I/O, performance and power. These estimations are performed by *light weight* synthesis estimators. These estimators have to be light weight because several thousand partition options may be examined. However, being light and accurate at the same time is very difficult. Sophisticated estimation techniques are used to alleviate this difficulty [26, 44, 22]. Behaviorally partitioned system may use more gates, since hardware is not shared between partitions. However, since RTL partitions are I/O dominated, the RTL partitions do not tend to under utilize the device. Thus, this increase in gates is not much of a concern.

The RC research community has invested several efforts into multi-FPGA Partitioning [57, 70, 35, 82, 51, 60, 11, 55]. However almost all of these have been post HLS partitioning

approaches. Chan et al. [57] partition with the aim of producing routable sub-circuits using a pre-partition routability prediction mechanism. Sawkar and Thomas [55] present a set cover based approach for minimizing the delay of the partitioned design. Limited logic duplication is used to minimize the number of chip-crossings on each circuit path. Bi-partition orderings are studied by Hauck and Borriello [70] to minimize critical bottlenecks during inter-FPGA routing. Woo [51], Kuznar [60], and Haung [11] primarily limit their partitioners to handle device area and pin constraints. A library of FPGAs is available and the objective is to minimize device cost and interconnect complexity [11, 60]. Functional replication techniques have been used [60] to minimize the cut size. Neogi and Sechen [35] presents a rectilinear partitioning algorithm to handle timing constraints for a specific multi-FPGA system. Fang and Wu [82] present a hierarchical partitioning approach, integrated with RTL/logic synthesis.

Behavioral partitioning has been promoted by several system level synthesis groups [65, 37, 21, 48, 22, 78, 80]. In this chapter the behavioral partitioning approach was chosen for RCs due to the drawbacks of structural, as mentioned above, and due to several studies that led to the decision [48, 38, 61, 79]. We present two integrated partitioning and synthesis methodologies RCs. In both approaches, we show that the physical memory on the RC can be effectively used to alleviate the pin-out and inter-FPGA interconnection bottle-neck. First, we present a data flow graph (DFG) partitioner. Following this a coarser block-level partitioner is presented. The block-level partitioner is integrated with a dynamic design space exploration engine. This chapter presents a fully automated framework for behavioral partitioning of a DFG and a CDFG, with appropriate estimation and exploration techniques such that the RC resources are effectively utilized. In addition, the chapter also provides a detailed summary of advantages and disadvantages of both partitioning approaches.

Various aspects of the partitioning problem presented in the chapter and the hardware area/performance estimation techniques bear similarities to the research in the area of hardware-software codesign [64, 58, 52, 23]. There have been several approaches to solve the problem of hardware-software partitioning for a range of granularity [65, 58, 30, 33, 13, 53, 76]. Fully automatic partitioners have been in existence for quite some time now [65, 58, 33]. Gupta and De Micheli [65] start with an all hardware solution and iteratively move one task at a time to software until no further improvement is possible. Ernst and Henkel [58] on the contrary follow a software oriented approach which starts with an all software solution and uses a simulated annealing partitioning engine. Hou and Wolf [33] proposed a process level partitioning heuristic based on hierarchical clustering. Eles [53] performs a performance guided partitioning based on simulated annealing and Tabu search. Thomas [13] presents a coarse grain partitioning methodology at a functional level. The RC partitioning presented in this work does not have a software estimation component. However, the communication model and the resource availability (both for communication between partitions and hardware logic) is well defined and performance overheads can be accurately computed within clock-cycle accuracy. The challenge is to dynamically explore the hardware design space and efficiently use the available communication resources inorder to generate the optimal design that satisfies resource constraints. It is typical of an RC environment to have a host desk-top computer interacting with a FPGA based RC . In such cases, RC hardware partitioning can follow functional level hardware-software partitioning.

The chapter is organized as follows. In the following section the DFG and block graph specification models are presented. Section 4.3 presents the target RC architecture model. Sections 4.4 and 4.5 present in detail the data flow graph and block graph partitioning methodologies, experimental results, and observations. Our conclusions are presented in the final

section.

4.2 Input Specification Models

In this section we formally present the two specification models that are used this chapter – data flow graph, and behavioral block graph.

4.2.1 Specification for Fine-grained Partitioning

Several digital signal processing (DSP) and image processing applications can be expressed as simple graphs that have pure data flow or minimal control flow. Discrete cosine transforms (DCT), Fast Fourier transforms (FFT), image filtering, and Jacobi transforms are some widely used applications that can be expressed by data flow graphs.

The input to be partitioned is an acyclic graph whose nodes are the operations to be partitioned across the FPGAs of the RC and the edges represent the data flow. The formal definition is as follows.

Definition 4.1 *A data flow graph (***DFG***) is a directed acyclic graph,* $\mathcal{G} = (V, E, I, O)$. *$V$ is the set of nodes representing the operations and E is the set of directed hyper-edges corresponding to the data flow dependencies. I is the set of primary inputs and O is the set of primary outputs. Following are the attributes related to nodes and edges.*

- *For each node* $v \in V$: $area_v$ *: Area of the nodes in CLBs (Configurable Logic Blocks).* i_v *: Number of input wires feeding* v. o_v *: Number of output wires fanning out* v. $level_v$ *: Level number of the node* v*, or its schedule time-step. For a valid DFG,*
  $v_i, v_j \in V \wedge v_i \rightsquigarrow v_j \Rightarrow 0 \leq level_{v_i} < level_{v_j}$,
  Here, the symbol $\rightsquigarrow$ *denotes a directed path.*
- *For each edge* $e \in E$: $source_e$ *: Source node or the primary input, in* I*, that drives the hyper-edge* e. $sink_e$ *: Set of nodes and primary outputs that are driven by the hyper-edge* e. $width_e$ *: Bit-width of the edge.* $mode_e$ *: Data-transfer mode for the edge* e*, if it cuts partition boundaries. If* $mode_e = MEM$*, then data-transfer is through memory transfer, else* $mode_e = WIRE$ *and data-transfer is through the interconnection-network of the RC.*

$\mathcal{G}$ is a scheduled data flow graph with well defined modes for data transfer across partitions. All primary inputs to the DFG must be available when the DFG starts execution (at time-step 0). Primary outputs are available in the next time-step after they are computed.

Figure 4.1 shows the DFG for a 8x8 vector product equation of the form $z = a_1.b_1 + a_2.b_2 + \cdots + a_8.b_8$. The DFG is scheduled in 5 time-steps. The level of an operation is the time step (level) at which it is scheduled. For instance, in Figure 4.1, 4 operations have the level attribute value of zero. All primary inputs to the DFG are fetched from the memory and all primary outputs must be stored in memory. All internal nets by default are *MEM* mode unless explicitly specified as *WIRE*. In Figure 4.1, two edges have *WIRE* mode, meaning, if these nets are cut during partitioning, then they must be wired using the interconnect resources on the board.

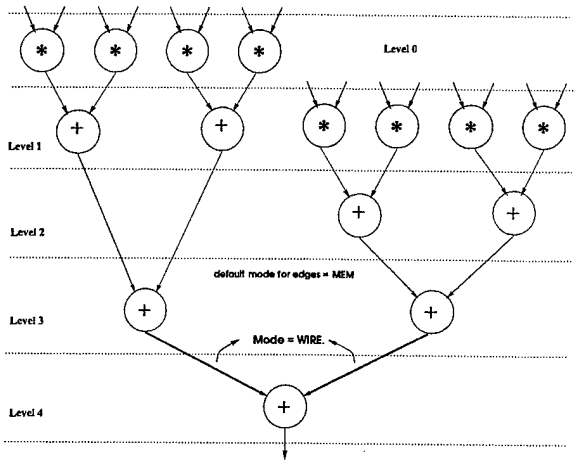


Figure 4.1: **Vprod: DFG for 8x8 vector product: Example**

4.2.2 Specification for Coarse-grained Partitioning

The *behavioral block graph* in essence is a Control Data Flow Graph (CDFG), where the blocks in the graph capture the data flow in the design and the edges across blocks capture both control flow and data transfers. The block graph is extracted from the behavioral specification of the design and is used as the intermediate format for the SPARCS synthesis tool called Asserta HLS [50, 49]. Here is a more formal definition of the BBG.

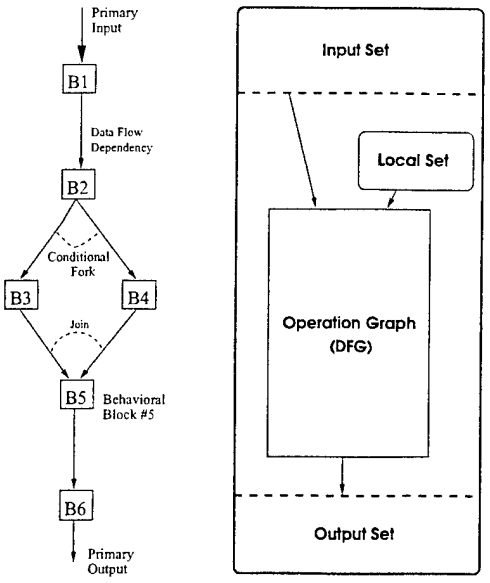
Definition 4.2 *A BBG is a directed graph,* $\mathcal{G} = (\mathcal{B}, \mathcal{C}_e, \mathcal{D}_e, PI, PO)$*, where*

$\mathcal{B} = \{b_1, b_2, \cdots, b_K\}$ *is the set of behavioral blocks in the design. PI and PO are the primary input and primary output ports of the design.*

$\mathcal{C}_e = \{CE_1, CE_2, \cdots, CE_N\}$ *is the set of control dependencies in the BBG. Each control edge,* $CE_i = < b_{s_i}, B_{d_i}, f_i > \in \mathcal{C}_e$*, denotes control dependency from BB* b_{s_i} *to the set of BBs in* B_{d_i}*. The control transfer from block* b_{s_i} *to one of the blocks in* B_{d_i} *happens conditionally based on the value that* b_{s_i} *sets on the flag* f_i*. Informally, each control edge is a multi terminal edge from a source block to multiple destination blocks, where the control transfer from source to one of the destination blocks occurs conditionally. Data transfer takes place from the output port of a block to the input port of another.*

$\mathcal{D}_e = \{DE_1, DE_2, \cdots, DE_M\}$ *is the set of data dependencies in the BBG. Each data transfer edge,* $DE_i = < b_{s_i}, B_{d_i}, w_i > \in \mathcal{D}_e$*, denotes multi-terminal net with BB* b_{s_i} *as the source and the set of BBs in* B_{d_i} *are the destination.* w_i *denotes the width of the data transfer edge. The source of the data-transfer may be a primary input port in PI and similarly one or more destinations may be primary output ports in PO. The variable* $mode_i$ *denotes the data-transfer mode (Definition 4.1) of* DE_i*.*

Each BB $b_i \in \mathcal{B}$ *is modeled as a four tuple,* $b_i = < G_i, I_i, O_i, F_i >$ *where,* G_i *is the DFG that represents the behavior of the block* b_i*.* G_i *follows the data flow graph semantics, Definition 4.1.* I_i *and* O_i *are the input and output ports of the behavior block* b_i*. The input ports of the block may be connected to one of the primary inputs (PI) or to the output ports of other blocks. Similarly ports in* O_i *may be connected to primary outputs (PO) and/or to inputs ports of other blocks. The port connectivity is also captured by the data dependency set,* $\mathcal{D}_e$*.*



(a) Example Block Graph (b) Behavioral Block

Figure 4.2: **Example of block graph, extracted from BBIF**

Figure 4.2(a) shows a block graph with 6 behavioral blocks. Figure 4.2(b) shows the structure of a single block. Block B1 performs operations on the primary input and the control is transfered to B2. On completion of B2, the control *conditionally* transfers to block B3 or B4. B5 executes next when either of B3 or B4 finishes and finally B6 is executed. *The block graph model permits only a single thread of control. Exactly one block is active at any time during the execution of the design.* This allows a *complete sharing of resources across blocks.* The block graph model allows operation-level parallelism within the behavioral blocks. Control constructs such as *if then else*, *case* and *while loops* in BBIF can easily be translated into control flow in the block graph [50, 49]. Figure 4.3 shows the BBG for the two dimensional FFT benchmark. The graph has 18 blocks and 25 edges. Notice that there are two loops in the graph.

4.3 Target RC Model

We consider a multi-FPGA RC architecture that has multiple FPGAs sharing a single physical memory. The FPGAs are interconnected by a fixed interconnection network and all the FPGAs can access the memory through a shared memory bus. Figure 4.4 shows the RC architecture model that is considered.

Formally, the RC has K FPGAs, $\mathcal{F} = \{f_1, f_2, \cdots, f_K\}$ that share a physical memory M. For $f \in \mathcal{F}$, $Area_f$ is the area of the FPGA in terms of the available CLBs. We define *conn* to be the connectivity matrix, where for $1 \leq i < j \leq K$, $conn_{ij}$ is the number of wires in the channel connecting FPGA -i and FPGA -j. The connectivity matrix is derived from the fixed interconnection network.

The FPGAs can communicate data either through the shared memory or directly through *channels* (wires) in the interconnection network. For DFG and block graph partitioning, all primary design inputs are assumed to be present in the memory, and all primary outputs are

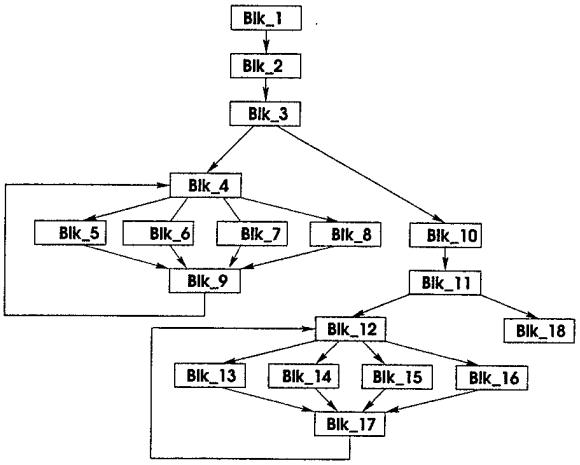


Figure 4.3: **Block Graph of 2D FFT**

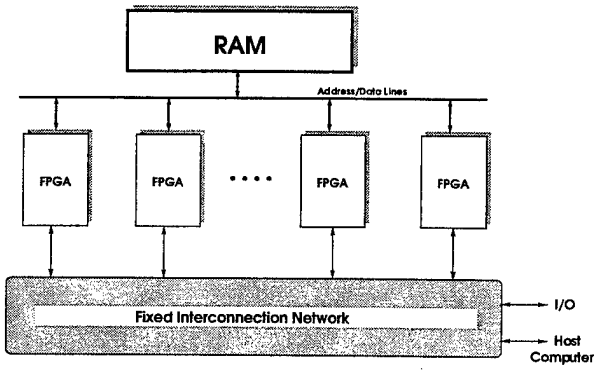


Figure 4.4: **The Reconfigurable Architecture Model**

written back to the memory. For DFG partitioning, when the internal edges in the DFG are cut, data communication is made through either memory or channels depending on their user-specified mode (Definition 4.1).

4.4 Data Flow Graph Partitioning

In this section we present the DFG partitioning methodology. First, we explain the partitioning and synthesis process. Following this the various details about the cost estimation and partition evaluation methodologies are presented. We briefly describe the iterative partitioning engine used. Elaborate experimental results are presented and the observations are summarized at the end of this section.

4.4.1 Partitioning and Synthesis Process for DFGs

The goal of the RC synthesis process is to partition and synthesize a behavioral specification and to *efficiently utilize* the underlying RC resources. The quality of the design is determined by the cost metrics, discussed later in Section 4.4.2. The primary goal of this process is to successfully obtain a partitioned design that satisfies all board-level constraints such as area, interconnect, and memory resources. The secondary, but an important factor is measured in terms of the performance (throughput) of the design.

Figure 4.5 shows the partitioning and synthesis design flow for DFG partitioning onto RCs. First the DFG is extracted from a behavioral specification in C [3] or VHDL (Very high speed integrated circuit Hardware Description Language) [27]. Design space exploration is performed to generate a *schedule* [10] for the DFG that is suitable for the underlying RC. The scheduled DFG is then passed as an input to the partitioner. The DFG partitioner generates multiple control data flow graphs (CDFGs). Each CDFG is synthesized for an individual FPGA on the RC. The *control* structures in the CDFG are a simple synchronization mechanism between the multiple communicating DFGs.

Any iterative partitioning engine such as simulated annealing (SA) [71], genetic algorithm (GA) [31], or Fiduccia-Mattheyses (FM) may be used by the partitioner. The most crucial component of the partitioner that determines its convergence is the *partition cost evaluator*. The evaluator estimates the cost and performance of the contemplated partition against the target RC model, as presented in Section 4.4.2.

Following partitioning, we generate a block graph (CDFG) for each partition segment of the original DFG. Each block graph is then individually synthesized to a RTL implementation by the Asserta [50] HLS tool. Asserta is a formally asserted high-level synthesis system, that can produce RTL designs for any user specified RTL component library. The RTL designs produced by Asserta have two components. The first component is a simple FSM (finite state machine) that acts as the controller and the second component is data path of components from the RTL component library. The synthesized designs are translated to VHDL, integrated together with the RC VHDL templates, usually provided by the RC vendor. The integration process involves connecting the design I/O to the pads in the RC template and generating additional glue logic, if necessary. Commercial VHDL simulators are used to simulate the partitioned design and verify timing and functionality of the partitioned design.

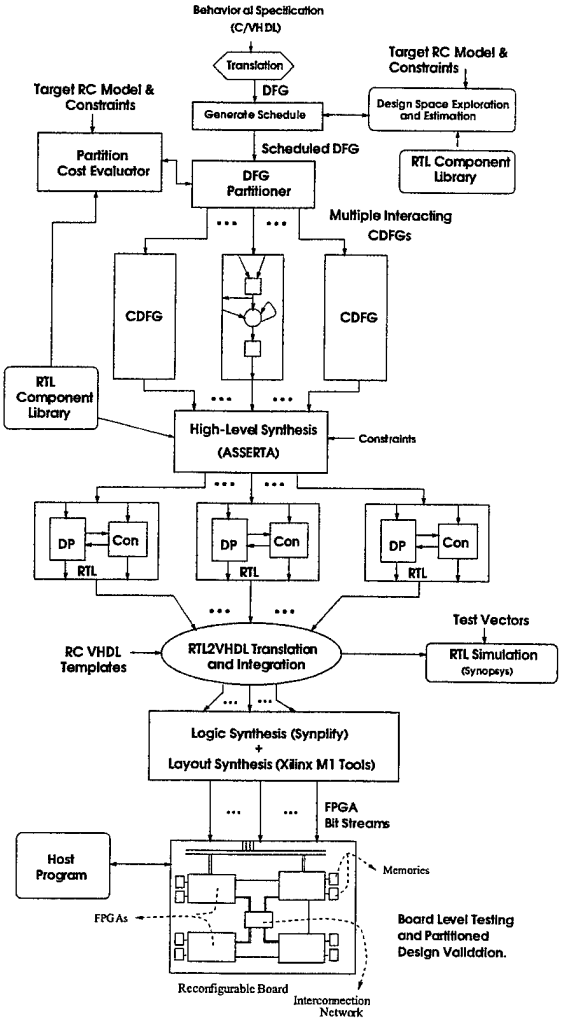


Figure 4.5: **RC Partitioning and Synthesis process for DFGs**

After successful RTL simulation, logic and layout synthesis is performed on each design that is mapped to an FPGA. The template host program is accordingly modified to setup the memory for inputs and provide the necessary signals to start and recognize the finish of the design on the RC. Finally the design is *downloaded* on the RC and board-level testing is performed. In practice, layout synthesis may fail during place and route. The information is fed back to the partitioner, respective constraints are tightened and the design process is carried out again. Due to this time expensive design cycle, the cost evaluators are carefully fine tuned so that such failures, late in the design process, can be minimized.

4.4.2 Partition Cost Evaluation for DFG Partitioning

Partitions are evaluated based on their architectural constraint-satisfaction and how well the performance is optimized. For DFG partitioning we consider number and area of the FPGAs and the inter-FPGA interconnection resources to be the constraints on the RC. The RC constraints are available from the RC architecture model (Figure 4.5). The optimization goal for the partitioner is to minimize the latency of the design. Before presenting the combined cost function, the area, interconnect and latency estimation techniques are presented.

Area Estimation

The area of each partition segment s is constrained by the number of CLBs available on the FPGA to which it is mapped. In the case of ASIC design both computation resources (ALU components such as, adders and subtracters) and storage resources (registers) are considered alike because both occupy silicon area. However, in FPGAs, the LUT (Loop Up Table) based function generators in the CLBs provide the computation resources while the flip-flops in the CLBs provide the storage resources. Hence in the case of FPGAs, computation area estimation and storage area estimation (register estimation) must be performed separately and individual constraint-satisfaction checked.

The total area of a segment is composed of the component area, and multiplexor areas due to component and register sharing. The area of each component is available from the RTL library. Multiplexor area characterization for Xilinx 4000 series FPGAs [43] shows that a four input multiplexor needs a unit CLB resource and the area increases linearly with the input size.

During synthesis, all inputs and outputs of every node in the DFG are stored in a register. Trivially if every input and output bit of a node is stored in a flip-flop, then the storage resource required is a function of sum of the I/O bits of all nodes in the partition segment. Approximately, for every two 4-bit registers that are shared, a unit CLB cost will be incurred for multiplexing. We use empirical formulas to compute the register and multiplexor area costs [77] based on an expected register sharing behavior.

Let $s_1, s_2 \cdots s_K$ be the K segments to be mapped onto the K FPGAs on the RC (Section 4.3). Then, we define the *AreaPenalty* of the partition as,

$$AreaPenalty = \sum_{i=0}^{K} \frac{\Delta A_i}{Area_{f_i}} \tag{4.1}$$

where, $\Delta A_i = \begin{cases} 0 & if\ Area(s_i) \leq Area_{f_i} \\ Area(s_i) - Area_{f_i} & otherwise \end{cases}$

$Area(s_i)$ is the estimated area of the segment s_i and $Area_{f_i}$ (Section 4.3) is the area of the FPGA to which the segment is mapped. Notice that no negative penalties are assigned to segments with areas lesser than the FPGA area. More importantly we normalize the area violations to the amount of area available. This is important when partitions are evaluated for multiple conflicting constraints. A value of zero for the *AreaPenalty* implies that the partition does not violate area constraint. Negative area penalties are avoided because the goal is to generate the performance-optimal design that does not violate resource constraints for a fixed RC. Negative penalties needlessly avoid search spaces thereby increasing the chances of being held at a locally optimal solutions.

Interconnect Estimation

The inter-FPGA interconnection resource constraints are derived from the connectivity matrix (*conn*) that is part of the RC specification model (Section 4.3). Let $s_1, s_2 \cdots s_K$ be the K partition segments. Each edge e in the original DFG contributes a wire requirement of size $width_e$, if $mode_e$ is $Wire$ and edge e communicates between two different segments s_i and s_j. Another component that adds to the interconnection cost is the routing of synchronization lines to the communicate end of a block execution.

Let $s_1, s_2 \cdots s_K$ be the K segments to be mapped onto the K FPGAs on the RC (Section 4.3). For $1 \leq i < j \leq K$, W_{ij} is the number of wires required between partition segments s_i and s_j (as computed by the function $Estimate\_Wires(s_i, s_j)$. Then, we define the *InterconectPenalty* of the partition as,

$$InterconnectPenalty = \sum_{1 \leq i < j \leq K} \frac{\Delta I_{ij}}{conn_{ij}} \tag{4.2}$$

where, $\Delta I_{ij} = \begin{cases} 0 & if\ W_{ij} \leq conn_{ij} \\ W_{ij} - conn_{ij} & otherwise \end{cases}$

Notice that similar to the *AreaPenalty* computation (Section 4.4.2), no negative penalty is assigned for constraint satisfying solutions. Again, the interconnect penalty is normalized with respect to the number of channels available on the RC for inter-FPGA routing.

Latency Estimation and Memory Utilization

A partition solution of a DFG is constraint-satisfying if both area and interconnect penalties are zero. When multiple constraint satisfying partitions are obtained, the partitioner selects the solution with the least design latency. Latency is the the total number of c-steps (clocks) to process a single set of inputs to the DFG. If the input specification were not partitioned but implemented on a single FPGA the total latency will be the schedule length of the DFG plus the time for reading primary inputs from the memory and writing primary outputs back to the memory.

In the case of a partitioned DFG, additional clocks are spent in data transfer across partitions and in synchronization of block execution across multiple segments. The latency of a partitioned

design is composed of four components.

1. DFG schedule length : This is the number of time steps in the scheduled DFG. This is equal to the maximum of the level numbers of all nodes in the DFG. Formally, schedule length = Max($\forall v \in V : level_v$), refer Definition 4.1.

2. I/O latency : Primary inputs and outputs are stored in the memory. Since we consider a shared memory RC model, the I/O latency is a linear function of the number of primary inputs and outputs of the DFG. If each memory read and write operation takes r and w cycles then the I/O latency for a DFG $\mathcal{G} = (V, E, I, O)$ is $|I|.r + |O|.w$. For the Wildforce [84] architecture, r is 3 cycles and w is 1 cycle.

3. Data transfer latency : When edges in the DFG cut segment boundaries, the data transfer between FPGAs may be either through the shared memory or through the channels in the interconnection network. Each data transfer through memory will consume $r + w$ cycles, and each wired data transfer will consume two cycles, assuming unit cycle for port read and write. If there are M data transfers through memory and W wired data transfers then the data transfer latency is $(r + w).M + 2.W$.

4. Synchronization latency: Insertion of synchronization blocks introduces extra clock ticks to write the done signal and recognize it. The synchronization latency is the number of synchronization blocks in the design multiplied by a constant factor. The value of the constant depends on the synthesis process. The HLS tool used (Asserta [50]) in our partitioning environment (Figure 4.5) consumes 4 extra cycles for every synchronization block.

The total latency is the sum of the four components discussed above. As seen from the above discussion, the latency of the the design can be accurately computed and the time to compute linearly increases with the number of edges in the DFG.

4.4.3 Partitioning Engine for DFG Partitioning

We use a GA (Genetic Algorithm) [31, 12] based partition engine to perform the DFG partitioning. GAs capture the solution in a structural representation and the convergence is based on genetic operations - selection, crossover and mutation. The genetic operators work on a population of solutions, also called generation. We now present the details of the adaptation of GA for DFG partitioning. We model the partitioning problem as a simple integer-coded genetic algorithm. Each partition solution is represented as an integer array whose length is equal to the number of nodes in the DFG and each location of the array has a value between 1 and the number of partition segments K. The selection operator probabilistically selects highly fit solutions in the current generation. The GA uses uniform crossover operation [39]. A mutation operator randomly changes the integer values in the integer arrays. The population size varies between 100 and 200. Selection percentage is set to 20% and mutation probability is 0.10.

The convergence of the GA is primarily dependent on the fitness computation function.
$fitness(x) =$

$$\frac{1}{1 + AreaPenalty(x) + InterconnectPenalty(x)}. \tag{4.3}$$

Table 4.1: **Design Data for DFG Partitioning**

Example Name	Num Nodes	Num Edges	Area (CLBs)	Latency (c-steps)
Vprod	15	14	258	55
StatFn	23	22	187	33
Reverb	22	24	1424	34
FIR	23	22	1044	81
Elliptic	36	49	1176	48
FFT-1D	40	16	736	74
FFT-2D	88	112	1652	80
MatMult	112	96	1905	117
DCT4x4	224	256	8200	166
DCT8x8	1929	2304	13999	835

Equation 4.3 shows how fitness of a partition, x is computed. *AreaPenalty* and *InterconnectPenalty* are computed as presented in Sections 4.4.2. Thus, when both area and interconnect penalties are zero, the fitness of the solution is 1.

The selection operator first *sorts* the solutions in the current population in decreasing order of *quality* (Equation 4.4). For two solutions x and y, the solution with

$quality(x) > quality(y) \iff$

$$(f_x > f_y) \bigvee((f_x = f_y) \wedge (lat_x < lat_y)) \tag{4.4}$$

where, f_x and lat_x are the fitness and latency values for the solution x.

The selection operator selects 20% of the population, with a high probability of selecting solutions lower (high quality solutions) in the sorted array. Selecting the lower 20% of the sorted array may make the GA converge too fast. The crossover operator also selects good quality solutions for crossover.

4.4.4 Experimental Results for DFG Partitioning

In this section we present the results of partitioning several DFG benchmarks of varying sizes onto multi-FPGA RC architectures. The genetic partitioning engine and the estimation algorithms are implemented in C++ and all results are reported for a two processor Sun UltraSparc workstation running at 296Mhz with 384MB RAM.

Table 4.1 shows the various designs that are partitioned. All examples were first written in straight-line (i.e, no loops or conditionals) C [3] and translated into the DFG format. The input operators are nibble sized.

- The **Vprod** DFG is shown in Figure 4.1. Vprod has 15 nodes and 14 edges (number of edges does not include the primary input/output connections). Column 4 of Table 4.1 gives

the area required to implement the design on a *single* FPGA. Column 5 is the latency of a single FPGA implementation. This latency is the sum of DFG schedule length and the I/O latency (Section 4.4.2). Column 5 also gives the lower bound on the latency that can be achieved by any multi-partition implementation of the corresponding DFG.

- **StatFn** computes a statistical function of an array of 8 nibble-sized inputs. It first finds the mean, then computes the deviation from the mean of each input. It then outputs the mean and the sum of products of the adjacent odd and even values in the deviation array. The design has 23 nodes and 22 edges.

- **Reverb, FIR, Elliptic, FFT-1D** are computationally small designs (number of nodes, edges ≤ 50) like Vprod and StatFn. *FIR* performs finite impulse response function on 16 inputs.

  Elliptic is the elliptic wave filter and *Reverb* is an implementation of the reverberation filter. *FFT1-D* does the one dimensional fast Fourier transform on a 4x4 matrix. The reason for increased area sizes of Reverb, FIR, FFT-1D and Elliptic is because the operator sizes are much larger (16-bit) than that of Vprod and StatFn.

- **FFT-2D** and **MatMult** are medium scale examples that have about 100 nodes and as many edges. The solution space of a partitioning problem is n^K where n is the number of nodes and K is the number of partition segments. Thus the solution space increases much rapidly with the increase in the size of the DFG. FFT-2D is the DFG of a two-dimensional fast Fourier transform and MatMult is a matrix multiplication operations of two 4x4 matrices.

- **DCT4x4** and **DCT8x8** are both DFG for two dimensional discrete cosine transform operation of 4x4 and 8x8 matrices. Both are large examples, the latter being a much larger example with around 2000 nodes and edges. Two dimensional DCT involves two matrix multiplication operations, one for each dimension.The DFG for these examples have several large 9-bit and 20-bit multipliers.

We partition the designs in Table 4.1 for the Wildforce family of architectures. Wildforce has 4 FPGAs, Wildchild has 8 and Wildfire has 16 FPGAs on the board. The FPGAs are Xilinx 4000 series FPGAs. For our experiments we consider the target RC to be one of the Wildforce family boards with all the FPGAs being the same Xilinx device. The Xilinx FPGAs we allow are XC4005 (196 CLBs), XC4013 (576 CLBs), XC4025 (1024 CLBs), XC4036 (1296 CLBs) and XC4085 (3136 CLBs). One of the memory devices on the board acts as the shared memory and a common memory bus (address, data and read/write control) is routed through all the FPGAs.

Only limited wires are available for data transfer across FPGAs. If the inter-FPGA interconnect resource is unavailable, the partitioner *automatically* transfers data through the shared memory.

Table 4.2 shows the results of partitioning the DFG designs presented in Table 4.1. Column 2 is the type of RC architecture and the FPGA device that is used. The designs Vprod, and StatFn are targeted to a board with 2 XC4005 devices and MatMult is targeted to an RC with two XC4025s. The rest of the designs are partitioned onto a Wildforce family board.

The DFGs are scheduled to match the constraints posed by the target RC. For the smaller examples ($Vprod \cdots MatMult$) in Table 4.1, the area and the latency correspond to the fastest (ASAP) schedule of the graph. The DFG for *DCT4x4* and *DCT8x8* have several multipliers of very large sizes (>20 bits wide) therefore the ASAP schedule was infeasible. In fact the ASAP

area estimates for *DCT4x4* is ≈40000 CLBs and for *DCT8x8* the estimate is ≈150000 CLBs. Hence a highly serial (slow) schedule, with aggressive sharing of operators, was generated for these two examples. The area and latency estimates for DCT in Table 4.1 corresponds to the slow schedule.

Column 3 of Table 4.2 shows the estimated areas of all the partition segments. The number of segments is equal to the number of FPGA devices on the board. Notice that some of the segments may be empty (*FIR, Ellip, FFT1D*). In fact as the number of segments increases, the latency of the design tends to decrease. This is illustrated by the results for DCT4x4 example. We target DCT4x4 to Wildforce (4 segments), Wildchild (8 segments) and Wildfire (16 segments). The size of the FPGA device decreases with the number of FPGAs on the board. We see that the design executes fastest on Wildforce architecture, followed by Wildchild then the Wildfire. The latency of an unpartitioned design (reported in Table 4.1), provides a lower bound on the design latency. Column 5 of Table 4.2 reports the latency of the partitioned design. Notice that the more the number of partition segments, the greater is the deviation from the lower bound.

The estimated area of the unpartitioned DFG (as in Table 4.1 is the lower bound on the total area. Column 4 of Table 4.2 reports the total area of the partitioned designs. We infer from the results that limited larger FPGAs is a better alternative than several smaller FPGAs. The run-times for the genetic algorithm rapidly increases with the size of the DFG. The last column in Table 4.2 reports the execution time to complete 1000 generations of the genetic algorithm.

For a few examples (Table 4.3), we performed logic and layout synthesis and compared our estimated area and performance measures against the actual values after synthesis. The designs Vprod and StatFn were successfully implemented and tested on the Wildforce [84] board. Table 4.3 shows estimated and actual areas after synthesis of the partition segments for each design. Similarly, the estimated and actual latency (obtained from RC VHDL template simulation) of the partitioned design is reported. We observe a very small deviation in the latency computation. The deviation in area estimates and actual area is due to the approximations made for interconnect and the controller area. However the error margin is less than 20%, making the estimation process reliable.

4.4.5 Observations and Summary for DFG Partitioning

We presented a simple and efficient methodology for partitioning data-flow graphs onto multi-FPGA shared memory RC architectures. A fully automatic design flow from a behavioral specification in C, or VHDL is accomplished. There are various advantages of this process.

- For small designs, it is straight forward to specify the design as a DFG.
- The user is oblivious to the underlying RC architecture. The specification is implementation independent. The data transfers and the DFG I/O are automatically mapped to the memory or the inter-FPGA wires automatically.
- Given a scheduled DFG, area and latency of the partitioned design can be estimated with reasonable accuracy.
- The partitioned implementation exploits the inherent operator level parallelism in the DFG. Multiple operations in the same and different FPGAs may be active at the same time.

- Typically, pin-outs and inter-FPGA routing resources are one of the primary bottlenecks to implement large designs on RCs. Our DFG partitioning methodology is an approach to minimize this bottleneck by seamlessly transferring inter-FPGA communication through the memory. This is done at the cost of increased latency of the design. However, *if not for this, most of the useful designs tend to be infeasible.*

- The partitioned design model is simple and closely resembles the original DFG in its structure. The simple partitioning process poses very limited scope for error when generating the partitioned design. Moreover verification of the partitioned design is not difficult.

The advantages of DFG partitioning come with several disadvantages too.

- The design space exploration process is not fully integrated with the partitioning process. Since partitioning is done at a fine-grained level (operation-level partitioning), the problem size increases exponentially with the number of nodes. Design space exploration (i.e., determination of an efficient schedule of operations), during partition is extremely time consuming and is feasible only for very small examples. In the DFG partitioning methodology, design space exploration is performed a priori based on the constraints posed by the target RC.

- The lack of control structures, such as conditional branching and loops, makes it tedious to specify larger designs. For ease of specification, if control constructs such as loops were allowed, the following are the drawbacks: 1) the transformation process into a DFG may be complex and prone to errors. 2) verification of the specification against the final design will be much more difficult.

  The DFG model is insufficient to express designs with control flow, a traffic light controller, for instance. However DFG specification is useful for a restricted domain of data dominant applications.

- Due to operator level partitioning, the problem size is much larger than functional partitioning. For instance, partitioning a 4x4 Matrix multiplication DFG is several times faster than an 8x8 multiplication. This is not true about block level or function level partitioning. As seen from the results (Table 4.2), run-times for DCT8x8 design is *several days.*

We see a need for coarser grain partitioners that keep the partition problem size within reasonable limits. Integrated design space exploration during partitioning is possible with coarser grain partitioners but not during DFG partitioning. Control structures such as loop and conditionals enhance the applicability of the methodology. RCs with fewer and larger FPGAs are more useful than those with several smaller devices.

4.5 Block Graph Partitioning

In this section, we present the block graph (coarse-grained) partitioning methodology. We illustrate the advantages of integrated design space exploration during partitioning. The design space exploration engine and the various cost estimation techniques are described in adequate detail.

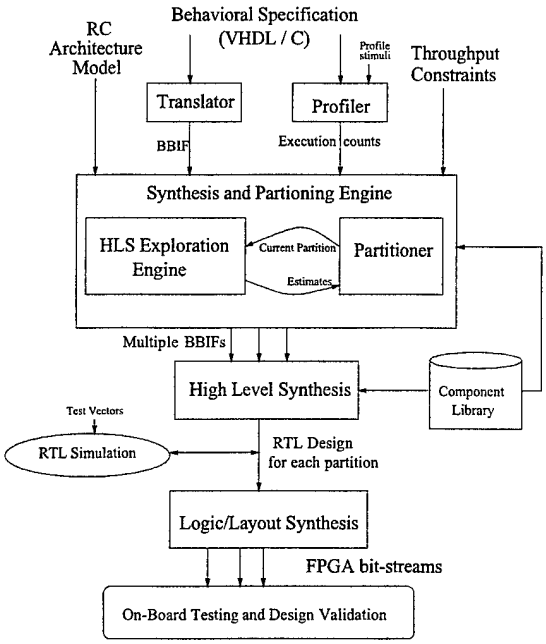


Figure 4.6: **Synthesis and Partitioning Environment for Block Graphs**

4.5.1 Partitioning and Synthesis Process for BBGs

Figure 4.6, shows our approach to integrated synthesis and partitioning. The design to be partitioned is specified in a high level specification language such as VHDL [27] or C [3]. The input translator converts the specification into an equivalent block graph specification (Section 4.2.2). A profiler computes the average execution count (AEC) and maximum execution count (MEC) for each behavioral block in the (block graph). AEC of a block is the average number of times the block is invoked per profile vector, averaged over a large set of profiling stimuli. MEC of a block is the maximum number of times the block is invoked during any single profile run. AEC values of blocks may be greater than one in the presence of loops and they may be fractional values less than one due to conditional invocation of blocks.

The core of the environment is the HLS exploration engine [69] integrated with an iterative partitioning engine. The unique feature of the exploration engine is that it views a partitioned model of the block graph and performs design space exploration to globally optimize the partitioned design. Traditionally design space exploration engines [59, 45, 14, 63, 9] do not consider partitioned design models for exploration. Instead area-time trade-off can be performed on each partition without the knowledge of other partitions. Thus each partition is locally optimized but global optimization of the partitioned design cannot be performed. We explain the multi-partition exploration engine in Section 4.5.2, and a more detailed description and analysis may be found in [69].

The partitioner has iterative partitioning engines such as the SA and the FM heuristics. The partitioners interact with the exploration engine through an application program interface (API). The API provides various useful functions that are used to efficiently perform design space exploration on a partitioned block graph. The various API functions and the modes of interaction between the partitioner and the exploration engine are presented in Sections 4.5.2 and 4.5.4.

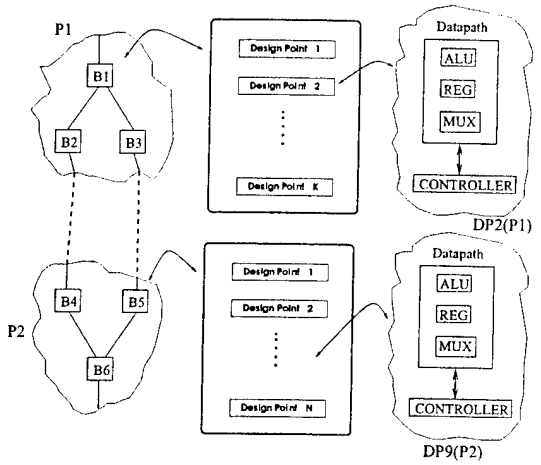


Figure 4.7: **Model for the Exploration Engine**

The result of partitioning is a collection of block graphs, one for each partition segment of the original block graph. Each partition segment is targeted to a single FPGA on the RC. HLS is performed on each partitioned block graph and a set of RTL designs are generated. Logic and layout synthesis is performed on the partitioned designs to generate the required configuration streams for the FPGA s. In our design process, the Asserta system [50] is used to perform HLS. Commercial simulation (Synopsys' VHDL simulator), logic synthesis (Synplicity's Synplify) and layout synthesis (Xilinx M1) tools are used.

4.5.2 Design Space Exploration Engine

The exploration engine [69] provides the framework to integrate any iterative partitioner, such as, SA, FM, and GA. The partitioners can invoke the exploration algorithm through a range of interface functions and get area and latency estimates about individual blocks, partition segments, or for the complete partitioned design itself. In addition to estimation, the exploration engine can efficiently explore the design space of the blocks (finding the best schedule for each partition segment), such that a constraint-satisfying solution that is optimal in terms of latency is produced.

Figure 4.7 shows the exploration model. On the left is the partitioned block graph. Each segment, s, has a set of design points associated with it, $\{DP_1^s, DP_2^s, \cdots DP_K^s\}$. The design points are shown in the center of the Figure 4.7. Each design point DP_i^s corresponds to a particular schedule of the segment s. A schedule of the segment s, is derived from the schedule of the individual blocks in s and the sharing information between the blocks. Given a design point for a segment s, various estimates about the RTL design corresponding to that design point can be made. These estimates include details about ALU, register, and multiplexor areas. In addition, the controller information such as the number of states, inputs and outputs are also available. The exploration engine also maintains the necessary information about sharing of resources within and across the blocks in any segment.

The overview of the exploration engine is presented in Figure 4.8. The inputs to the exploration are the BBIF corresponding to the *block graph* to be partitioned (with *blocks assigned to segments*), the *design latency constraint*, and the characterized RTL component library. The

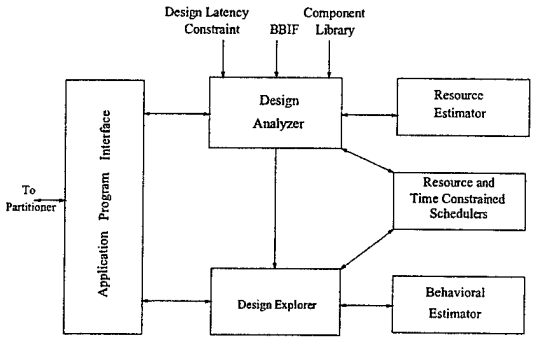


Figure 4.8: **Block Diagram of the HLS Exploration Engine**

exploration engine has three main components 1) The design analyzer 2) the design explorer and 3) the application program interface (API) to the partitioner. The exploration engine also has a resource estimator, fast time-constrained and resource-constrained schedulers and a behavioral estimator that can estimate the area and performance of the RTL design corresponding to a scheduled block graph.

The first call made by the partitioner to the exploration engine is to initialize the exploration engine and setup partition invariant information about the design. The *design analyzer* module does such initialization. Following are the phases of the design analyzer.

1. Initialization: Dependency graphs for each block and the entire control flow graph of the design is built.

2. Resource Estimation: For each block in the graph a *local resource set* is formed. The resource set contains components from the RTL library that are potentially required to implement the block. A *global* resource set for the entire graph is formed based on the individual local sets.

3. Design Latency Function: The function to compute the design latency is determined. Design latency is the number of clock cycles required for a single execution of the design. The design latency, $\mathcal{L}_d$, is the sum of the number of clocks spent in each block of the design. Formally,

$$\mathcal{L}_d \leftarrow \sum_{b \in \mathcal{B}} schedule\_length(b) \times EC(b) \tag{4.5}$$

   $EC(b)$ is the execution count of the block b. The execution count may either be set to the average execution count (AEC) or the maximum execution count (MEC) of the block. This choice is made by the user. Usually AEC is the default choice unless the latency constraints are very crucial and *must* not be violated. The user may choose to specify a specific latency computation function that is different from the above. The HLS exploration engine only generates schedules that satisfy the user given *design latency constraint*, i.e., $\mathcal{L}_d \leq$ design latency constraint.

4. Bounding Schedule Lengths: The fastest and the slowest schedule lengths for each block are computed. The fastest schedule length corresponds to the length of the ASAP [10] schedule

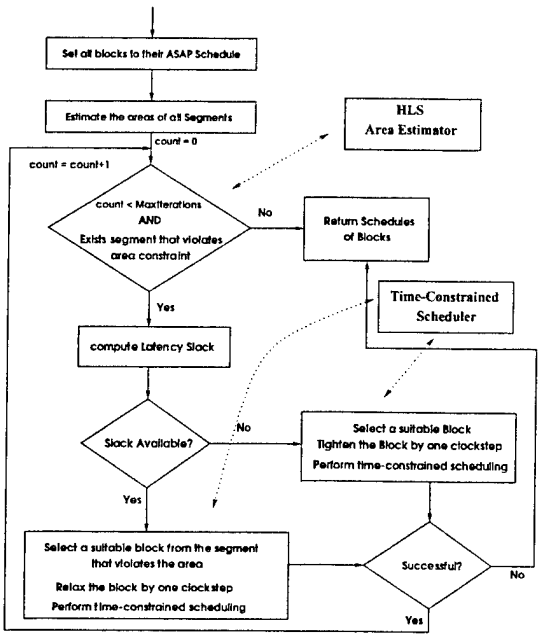


Figure 4.9: **Flow chart for the HLS Exploration Engine**

and the slowest schedule length is the length of the optimal schedule possible with the minimal resource bag (exactly one instance of each component in the local set). Based on the fastest and the slowest schedule lengths, the lower and upper bounds on the latency, $\mathcal{L}_d$ is computed. A *legal* design latency constraint must lie within these bounds.

5. Initial Schedule for Blocks: Initially, the design points for all blocks are set to the ASAP schedule of the block (Figure 4.9). Thus the initial design of the block graph *will not* violate any legal design latency constraint. However, one or more segments may violate the area constraint.

The working of the design exploration algorithm is explained through the flow chart in Figure 4.9. First all blocks in all partition segments are set to their respective ASAP schedules and areas of all segments estimated. If none of the segments violate the FPGA area constraints, then the algorithm returns the computed schedules, else, the exploration process begins. First the latency slack ($\mathcal{L}_{slack}$) available for the design is computed. The slack is the difference between the current design latency and the constraint, mathematically, $\mathcal{L}_{slack} = design\ latency\ constraint\ -\ \mathcal{L}_d$. The algorithm guarantees that given a legal latency constraint, $\mathcal{L}_{slack}$ is always ≥ 0.

If slack is available, a block (B_{relax}) is chosen from the segment that violates the area the most, and is relaxed. If slack is unavailable, a block ($B_{tighten}$) is chooses from the segment that violates the area the least and is tightened. Since the block graph is a single control thread, tightening any block in the design will minimize the design latency (also refer to definition of $\mathcal{L}_d$). The block B_{relax} is chosen such that relaxing it (i.e, increasing its scheduling length by one extra cycle), potentially causes the maximum decrease in resource requirement. Similarly the block for $B_{tighten}$ is chosen such that tightening (constraining its schedule by one less clock step) it has the least potential to increase in area of the segment.

The design space for all the blocks and in turn the design space for the partition segments are explored by this process of tightening and relaxing the schedules of the blocks. The algorithm

successfully exits when it finds a design that satisfies the area and latency constraint. Notice that since we start from the fastest design, the performance of the constraint satisfying design is automatically optimized. The algorithm returns the best available solution when an area constraint satisfying solution is be found. Notice that due to the nature of the algorithm, a latency constraint violating solution is never generated.

A combination of cone-based list scheduling (CBLS) [66] algorithm and the improved version of the Paulin and Knight's force-directed list scheduling (FDLS) [56, 74] algorithm is used to perform the time-constrained scheduling of the blocks. The same scheduler is used to estimate the areas for the entire segment and to generate the corresponding schedule. The time-constrained scheduler is shown in a shaded box in Figure 4.9. Another important component of the exploration engine is an area estimator (shaded box in Figure 4.9). The area estimator computes the number of CLBs required to implement each partition segment. The area of a segment is computed based on the following - the data path component areas, register area[1], multiplexor area due to ALU and register sharing, and finally the controller area.

Exploration Engine API Functions

The exploration engine provides the following four interface functions.

1. **ExploreDesign()**: Given the block graph, the binding of blocks to segments, and the area constraint on each segment, this function performs the algorithm in Figure 4.9, as explained above. This may be a time expensive algorithm because this involves iteratively rescheduling several blocks in the entire graph. The function returns the area estimates for all the segments based on the best solution obtained through exploration.

2. **ExploreSegments(S)**: This function is a restriction of the above function. Only the blocks in the partition segments $s \in S$ are explored. The schedules for other blocks remain the same. This function is considerably faster than ExploreDesign() when the number of segments is large and S is a small subset. After the exploration process, the segments in S are estimated for area and the values are returned.

3. **ExploreBlock(B)**: This function explores only the block B. Area estimation is performed for the segment to which B belongs and the value obtained is returned.

4. **EstimateDesign()**: This function does not do any exploration. Instead areas of all segments are estimated based on the current design points (schedules) of the blocks.

5. **EstimateMove(B, s_i, s_j)**: The binding of the block B is changed from segment s_i to segment s_j and the areas of two segments are re-estimated. This function does not perform any exploration. The function is very fast and extremely useful in the case of partitioners like FM [7] and SA [71], where the partitioner makes incremental moves by changing the segment binding of one node at a time.

4.5.3 Partition Cost Evaluation

The partition cost evaluation is similar to the evaluation criteria followed for DFG partitioning in Section 4.4.2. FPGA area and the inter-FPGA interconnect resources are the architectural constraints. The architectural constraints (Figure 4.6) are available from the RC architecture

[1]Register area is computed separately as the number of flip-flops in the target FPGA device.

model. As described in Section 4.5.2, the exploration engine can handle a *legal* design latency constraint (Figure 4.8). The exploration algorithm, as explained in the previous section, guarantees the satisfaction of latency constraint and also optimizes the design for latency when multiple area constraint satisfying solutions exist.

Area Estimation: Let $s_1, s_2 \cdots s_K$ be the K segments to be mapped onto the K FPGAs on the RC (Section 4.3). Then, we define the *AreaPenalty* of the partition as,

$$AreaPenalty = \sum_{i=0}^{K} \frac{\Delta A_i}{Area_{f_i}} \tag{4.6}$$

where, $\Delta A_i = \begin{cases} 0 & if\ Area(s_i) \leq Area_{f_i} \\ Area(s_i) - Area_{f_i} & otherwise \end{cases}$

$Area(s_i)$ is the estimated area of the segment s_i and $Area_{f_i}$ (Section 4.3) is the area of the FPGA to which the segment is mapped. The AreaPenalty is computed similar to the computation for DFG partitioning, Section 4.4.2. Here the estimated area of the segments ($Area(s)$) are obtained through the exploration and estimation functions provided by the API of the exploration engine.

Interconnect Estimation: The interconnect estimation procedure and the *InterconnectPenalty* computation is identical to the approach presented earlier for DFG partitioning, Section 4.4.2.

Latency Estimation: As mentioned above, latency of the partitioned design is posed as a constraint to the HLS exploration engine. The estimated latency of the partitioned RTL design is reported by the exploration engine. This value may be lesser than the constraint but is never greater.

The *fitness* and *quality* of partitions are computed based on Equations 4.3 and 4.4, as defined in Section 4.4.3.

4.5.4 Integration of Partitioning with HLS Exploration

We consider the simulated annealing algorithm for partitioning because it is most suitable for interaction with the interface provided by the exploration engine. Simulated annealing is more suitable for incremental estimation. The algorithm starts with an initial random partition, which we will synonymously refer to as the initial *configuration*. The partitioner moves from one configuration to another, typically making incremental moves. The incremental exploration and estimation functions provided by the HLS exploration engine can be efficiently used by the partitioner.

The partitioning engine communicates the initial configuration to the exploration engine. From then on, throughout the partitioning process, both the partitioning engine and the exploration engine maintain the same *current configuration*. As and when the partitioner changes its configuration by moving blocks across segments, the configuration in the exploration engine is changed accordingly. In addition to maintaining the configuration information, the HLS exploration engine, at any given time, maintains design space for all partition segments. For each block in the block graph, the exploration engine has a current design point (CDP) for the block ($\equiv$ current schedule for the block).

At the start of partitioning, for all blocks B_i, CDP_i is set to the ASAP schedule of the block. From then on, CDP_i is changed only when any of the exploration functions (*ExploreDesign()*,

Algorithm: `Simulated_Annealing`

```
1   [C_current = Random Initial Configuration]
2   [Evaluate C_current]
3   Temp = Initial Temperature
4   while (Temp > Final Temperature)
5      for (Iteration = 1 to ItersPerTemp)
6         C_new = Perturb(C_current)
7         [Evaluate C_new]
8         if (acceptable(C_new, C_current, Temp))
9            [C_current = C_new]
10        end if
11     end for
12     Temp = α * Temp
13     [Conditionally Perturb C_current]
14  end while
```

Figure 4.10: **Template Partitioning Algorithm for Simulated Annealing**

ExploreSegments(), or ExploreBlock()) changes the schedule of block B_i. We now present the template of the SA algorithms and its interaction with the exploration engine.

Figure 4.10 presents the template of the SA based partitioning algorithm. The statements that are boxed (statements 1, 2, 7, 9, and 13) are the stages in the algorithm when the partitioner interfaces with the exploration engine.

- *Statement 1:* SA creates a random initial configuration $C_{current}$. This initial configuration is communicated to the HLS exploration engine. The partitioner also invokes the *design analyzer* (Figure 4.8) to perform the initialization routines of the exploration engine, as mentioned in Section 4.5.2.

- *Statement 2:* The cost of $C_{current}$ is evaluated (Section 4.5.3). An interconnection estimator (Section 4.4.2) is invoked to compute the interconnect penalty. To compute the area penalties, the **ExploreDesign()** function is called to explore and generate the best design points (DP) for all blocks, for $C_{current}$. The exploration engine reports the estimates obtained for the RTL designs for the best DPs. It must be understood that the quality of the design space is dependent on the configuration. A set of DPs for blocks that are optimal with respect to one configuration may be poor for another.

  After computing the area and interconnect penalties, the fitness function in Equation 4.3 is used to compute the fitness of the configuration. The *quality* of the partition is used as the evaluation metric. The quality of partitions are evaluated based on the relation defined by Equation 4.4.

- *Statement 7:* At this stage the SA is looking at a neighborhood configuration to move to. In statement 6, the new configuration C_{new} is generated. Interconnect penalty is computed the same way as before. Ideally, invoking the ExploreDesign() function for C_{new} will produce the

optimal design space and the corresponding RTL estimates. However, ExploreDesign() is a time expensive function and more importantly, Statement 7 is within the inner loop (stmt 5 - 11) of the SA. To keep the execution time of the SA under control, the **EstimateDesign()** function is invoked[2].

The premise here is that, since the C_{new} is in the neighborhood of $C_{current}$, the optimal design space for $C_{current}$ is usually near-optimal, or good at the least, for C_{new}. If C_{new} is obtained from $C_{current}$ by moving exactly one block from its current segment to another, then the **EstimateMove(...)** function may be invoked. This provides additional speedup in comparison to the EstimateDesign() function.

- *Statement 9:* The current configuration in the partitioner is updated. Accordingly, the configuration within the exploration engine must be changed. The **ExploreDesign()** is called to create the best design space for the newly accepted configuration.

- *Statement 13:* This is a minor variation to the standard SA algorithm. If $C_{current}$ has not changed over several temperatures, then it is more than likely that SA is a local-minima. To get SA out of the local-minima, we perturb $C_{current}$ randomly[3]. The newly obtained configuration is communicated to the exploration engine and ExploreDesign() is invoked to create the best design space for the new configuration.

The template for the SA shows how a move based partitioning algorithm can be efficiently integrated with the exploration engine. The time expensive exploration and the fast estimation functions are appropriately used by the partitioner. The integration of design space exploration during the partitioning process is possible due to the reduced problem size of block-level partitioning in comparison to DFG partitioning.

4.5.5 Experimental Results for Block-Level Partitioning

In this section we present the results of block-level partitioning and design space exploration. We study the behavior of block-level partitioning for several benchmarks that were used for DFG partitioning (Table 4.1), with minor or no variation. We compare the block-level partitioning and synthesis methodology with the DFG synthesis and partitioning methodology presented in Section 4.4. The main difference is the presence of control in the specification and the reduction in the problem size of block graph partitioning in comparison with the DFG counterpart. This allows dynamic design space exploration during the process of partitioning. We implemented the simulated annealing (SA) based partitioning engine and the exploration engine discussed in Sections 4.5.4 and 4.5.2. The implementations are in C++ and all results are reported for a two processor Sun UltraSparc workstation running at 296Mhz and 384MB RAM.

The details of the SA based implementation are as follows. The initial solution was created randomly, and solutions were perturbed by changing the partition of one of the blocks in the design. The starting temperature was 30,000 and was cooled down very slowly, using a cooling factor of 0.999997, to a final temperature of 0.1. The SA was allowed to iterate 50 times at each temperature value. The fitness value of the partition is recomputed after each perturbation. If f_c

[2]In fact, we did experiment with ExploreDesign() function at this stage of the SA but run times were prohibitively high ($> 12hrs$) for even for relatively small examples, like FFT, Figure 4.3.

[3]At any time, SA stores of the best configuration obtained thus far. Hence, it is not a concern if the current configuration is indeed the global optima.

is the fitness of the current solution and f_p is the fitness of the solution after perturbation, then the improvement factor, IF is defined as $(f_p - f_c)/f_c$. Perturbed solutions with positive improvement are always accepted and those with negative improvement factor are accepted with a probability of $e^{IF/T*C}$, where T is the current temperature and C is a constant factor with a value of 0.000003. All constants were are results of tuning over a large set of runs.

Table 4.4 presents the details of the block graph benchmarks. All designs were first written in behavioral VHDL [27] and translated into BBIF (the internal format to store the block graph for partitioning and synthesis). Unlike the specification for DFG partitioning, loops and conditionals may be present in the VHDL specification of the design. Several of the benchmarks in Table 4.4, such as, Reverb, FIR, FFTs and DCTs, are *functionally identical*[4] to the examples used in Section 4.4.4. As mentioned in Section 4.4.4, the examples in Table 4.1 were translated from a straight-line[5] C [3] code. However the block graphs in Table 4.4 are translated from VHDL descriptions that may have loops, conditionals and case statements.

The VHDL descriptions were written in a manner favorable for block-level partitioning. Block boundaries in the VHDL specification usually occur at control expressions, such as loops and conditionals. Block separation also takes place at points where there are I/O reads or writes in the specification. The user has to intelligently write the VHDL specification. Larger, but fewer number of blocks, reduces the size of the block graph but on the flip-side may require larger FPGAs in the RC because blocks cannot be partitioned across partition boundaries. Smaller, but several blocks introduce two problems: 1) the size of the block graph increases and so will the time to partition; and more importantly, 2) the lower bound on the achievable design latency increases due to clocks spent on transfer between block. Moreover, blocks execute serially, hence parallelizing operations in different blocks is not possible even when the required resources are available.

Table 4.4 provides the following information about the benchmark block graphs. The first column is the name of the example. Column 2 shows the number of blocks in the block graph. Note that not all blocks in the block graph perform active computations. There may be several I/O blocks and simple condition evaluation blocks, depending on the nature of the VHDL specification. Column 3 is the number of data edges in the graph. The number of loops in the VHDL specification of the block graph is in Column 4. The table does not show details about the number of conditional expressions (*if-then-else* and *case-when*).

The $\mathcal{L}_{min}$ of the design is the lower bound on the achievable latency[6] of the design. This corresponds to the ASAP [10] schedule length of the design when implemented as a single partition. $\mathcal{L}_{max}$ is the *tight upper bound* on the latency (refer to Bounding Schedule Lengths, Section 4.5.2). $\mathcal{A}_{max}$ is the *maximum* area required by any implementation of the design that achieves $\mathcal{L}_{min}$. $\mathcal{A}_{min}$ is the *minimum* area required by any implementation of the design that is no slower than $\mathcal{L}_{max}$. Informally, these are *tight bounds* on the latency and area required for implementing the design. Their values are computed as follows.

- $\mathcal{L}_{min}$ is computed by performing an ASAP schedule of all blocks in the graph
- $\mathcal{L}_{max}$ is derived by performing a resource-constrained scheduling of the design where exactly one resource of each required type is available.

[4]Same input/output functionality and bit-widths. But the timing aspects may be different.
[5]No control constructs.
[6]Latency includes the time required to read/write primary inputs/outputs from and to the memory.

- $\mathcal{A}_{max}$ and $\mathcal{A}_{min}$ are produced by time-constrained scheduling of the design where the constraints on the schedule are $\mathcal{L}_{min}$ and $\mathcal{L}_{max}$ respectively.

Following is the description of the block graph benchmarks in Table 4.4.

- **Find**: *Find* is a control dominated design with 3 loops and several conditional evaluations. *Find* first sorts an array of 8 16-bit integers using the bubble sort algorithm. After the sort, it can take one 16-bit input per run and search for the existence of the input number in the sorted array using a binary search mechanism. Clearly control dominated examples such as *Find* cannot be written as DFGs. The *Find* block graph has 22 blocks and 30 edges. There are 3 loops and several conditionals (not mentioned in the table) in its VHDL specification. The minimum and maximum latency bounds of *Find* are the same (584 cycles) because one resource of each type is sufficient to achieve the ASAP schedule length.

- **ALU** is a small design that reads two 8-bit numbers and a 2-bit opcode and produces a 16-bit result. Depending on the opcode, the result is either the sum, difference or the sum-of-products of the two numbers. There is no appreciable difference in the lower and upper bound values of latency and area.

- **MeanVar** reads in 8 4-bit integers and produces the mean and the 11-bit variance as the result.

- **Reverb, FIR, Elliptic, FFT-1D** are functionally identical to the DFG graphs in Section 4.4.4. Their functional descriptions are also available in Section 4.4.4. Notice that FFT1D is implemented with one loop. In general, for these examples there is not a great difference between the $\mathcal{A}_{max}$ and the $\mathcal{A}_{min}$ values.

  The $\mathcal{L}_{min}$ value is comparable to the ASAP schedule length of the corresponding DFG, as reported in Table 4.1. The $\mathcal{L}_{min}$ values are always marginally more than the DFG's ASAP values. This is due to the additional clocks for block transfers and the potential parallelism that may be lost between operations in different blocks. The important values to observe in the table are the $\mathcal{A}_{min}$ and $\mathcal{A}_{max}$ values. Notice that both these values are much smaller than the area required for the ASAP schedule of the DFGs. In the case of the FFT-1D the difference is large. The reduction in area values is through the use of efficient schedulers [66, 74, 56].

- **FFT-2D, MatMult, DCT4x4** and **DCT8x8** are the larger (in terms of problem size, and more in terms of the area of the design) set of benchmarks. These are functionally identical to their DFG counterparts in Section 4.4.4. Notice that the difference between the $\mathcal{L}_{min}$ and $\mathcal{L}_{max}$ increases with increasing number of operations in the design. For DCT8x8, $\mathcal{L}_{max}$ is more than 3 times $\mathcal{L}_{min}$ (over 1000 clocks slower). Accordingly, we see a similar trend in the $\mathcal{A}_{min}$ and $\mathcal{A}_{max}$ values. For DCT8x8, $\mathcal{A}_{max}$ is about 3.5 times $\mathcal{A}_{min}$. The area estimates are comparable to the DFG area estimates in Table 4.1. As mentioned in Section 4.4.4, the DFG for the DCT benchmarks correspond to their *slowest* schedules. Hence for the DCT examples compare their $\mathcal{A}_{min}$ values with the areas reported in Table 4.1.

Similar to the results presented for DFG partitioning (Section 4.4.4), we try to partition the block graphs in Table 4.4 for the Wildforce [84] family of architectures. Wildforce has 4 FPGAs, Wildchild has 8 and Wildfire has 16 FPGAs on the board. The FPGAs are Xilinx 4000 series FPGAs. For our experiments we consider the target RC to be one of the Wildforce family boards

with all the FPGAs being the same Xilinx device. In order to effectively compare the DFG and block level partitioning we use the same FPGA devices that were used for DFG partitioning. The Wildforce architectures can host the following Xilinx FPGAs – XC4005 (100 CLBs), XC4005 (196 CLBs), XC4013 (576 CLBs), XC4025 (1024 CLBs), XC4036 (1296 CLBs) and XC4085 (3136 CLBs). One of the memory devices on the board acts as the shared memory and a common memory bus (address, data and read/write control) is routed through all the FPGAs.

Only limited wires are available for data transfer across FPGAs. During partition cost evaluation, edges in the block graph that cut segment boundaries are *automatically* converted to memory based data transfer. Accordingly a latency penalty is introduced. In our experimentation, we associate a latency penalty of 6 cycles (3 for memory read + 1 for memory write and 2 for synchronization) for each data transfer through memory. Thus a partition with a larger cut-set will have a larger latency.

Table 4.5 presents the results of partitioning the block graphs in Table 4.4. Column 2 is the type of RC architecture and the FPGA device that is used. The designs ALU, MeanVar, Find, and MatMult are targeted to a 2-FPGA RC board where the FPGAs are XC4003, XC4005, XC4013 and XC4025 respectively. Rest of the the designs are partitioned onto a Wildforce family RC board. To aid effective comparison between the partitioners, for the benchmarks common to Table 4.2 and Table 4.5, the same identical target architectures are chosen.

We analyze the results in Table 4.5 based on **Design Area**, **Design Latency** and partitioner **Run time**.

Design Area

The total design area for all benchmarks is close to, or is comparable to their corresponding $\mathcal{A}_{min}$ value in Table 4.4. This shows that the partitioner, with the aid of the design space exploration engine, converges to design configuration that minimizes duplication of resources (functional units and ALUs) in multiple FPGAs. We see that the total design area after behavioral partitioning is comparable to the total design area of the unpartitioned design.

For almost all benchmarks (DCT4x4 being the exception), the total estimated design area after block graph partitioning (Table 4.5) is less than the total estimated design area after DFG partitioning (Table 4.2). The design space exploration engine utilized the available area better and produced a faster design. For the largest design, DCT8x8, total area after block partitioning is much smaller than the design after DFG partitioning (22183 CLBs vs. 36556 CLBs).

For the FFT1D benchmark, block partitioning only required two XC4005 FPGAs while the DFG partitioner required two XC 4013s. The total design area of the FFT1D after block partitioning is 335 CLBs (this is fit on a single XC4013), while after DFG partitioning it is 736 CLBs (Tables 4.5, and 4.2). The estimated design latencies are comparable (91 and 109 clocks steps). The reason for the reduced area with comparable performance is due to the efficient design space exploration during the partitioning process.

The DFG partitioner failed to partition DCT8x8 onto a Wildchild board. It required Wildfire board with 16 XC4085s (the largest FPGA chip available). Table 4.5 shows two runs for the DCT8x8 benchmark. After the first run we noticed that the estimated areas of some of the partition segments were very close to the area of the FPGA (3196 CLBs). Hence we tightened the area constraint to 2900 and partitioned the design again. As expected, the latency of the design increased from 1752 cycles to 1806. Interestingly the maximum area of any segment reduced to 2900 but the total design area increased. This is because, as constraint is tightened, certain

blocks that shared resources with other blocks in the segment are forced to other segments resulting in new resources being instanced. This goes back to the observation made at the end of in Section 4.4.5 that fewer larger FPGAs are better than several smaller devices.

Design Latency

For most of the examples the design latency after partitioning (Table 4.5) is comparable to its $\mathcal{L}_{min}$ value. For most benchmarks, the estimated design latency of the block partitioned designs is better than or close to the latency of designs resulting from DFG partitioning (Compare Tables 4.5, and 4.2 for Reverb, FIR, Ellip, FFT2D and DCT4x4). FFT1D after block partitioning has a slower latency because the design is much smaller (in terms of number of CLBs) and hence is a slower implementation. In general DFGs are larger and have more nodes and edges when compared to equivalent block graphs. Due to this, the number of edges that cut partitioning boundaries in a design resulting from DFG partitioning tends to be more than block partitioning. As designs get larger, partitioning the DFGs gets more complex and the design latency is dominated by the data transfer component (Section 4.4.2. For this reason, the DFG partitioning for DCT8x8 produced a design with very high latency (about 14000 clock cycles) while block partitioning generates a design with a much lower latency 1800 clock cycles.

Partitioning Run times

For the smaller designs (ALU $\cdots$ FFT1D), partitioning run time is only a few seconds ($\leq$ 10 seconds). For FFT2D and MatMult the SA run time is between 20-25 seconds. For the DCT examples, the run times is about 5 minutes for DCT4x4 and about 2 hours for DCT8x8. In comparison to DFG partitioning, this is a tremendous improvement in partitioning run times (Table 4.2). For the DCT8x8 benchmark, the DFG partitioner did not converge even after a run time of 182 hours. These results show the feasibility of block graph partitioners to handle large designs. As we tighten the constraints for block partitioner (for example, by making FPGAs, on the RCs smaller), the block partitioning run times are bound to increase. However, based on the above results, it is highly likely that a good-quality solution will be produced in a reasonably small run time.

From our experiments, we see that block level partitioning is superior to DFG partitioning both in terms of quality of the partitioned design and also in terms of partitioning run time. The DFG partitioning methodology is useful only for relatively small designs that have no control flow. We performed logic and layout synthesis for the partitioned designs of ALU and MeanVar and successfully verified the estimates and the functionality of the synthesized designs.

4.5.6 Observations and Summary of the Block-Level Partitioning

We observe that the block level partitioner has the following advantages.

- Typically, block graphs can be modeled much smaller (using loops) than the equivalent (un-rolled) DFGs. The integrated design space exploration engine can generate an implementation of the design that is optimal with respect to the current partition configuration for the area constraints posed by the target RC.
- The RTL implementation model of the block graph is simple. Each partition segment is implemented as a single controller, single datapath design.
- Experiments show that the quality of the resulting partitioned designs are superior to that

generated from DFG partitioning. This is attributed to efficient dynamic exploration of possible schedules for the blocks in the design.

- Various implementation alternatives can be tried by varying the design area and latency constraints. Experiments and our experience with the partition and synthesis environment shows that the response of the partitioner to changes in user constraints is very intuitive and can be easily understood, and effectively used.

- Block graph model provides the user with the flexibility to specify very large designs. In the case of DFGs, the partitioning and synthesis complexity increases exponentially with the problem size. However, efficient modeling of design in terms of block graphs keeps the problem size under control. For example, the DCT8x8, a 1929 node DFG, design is efficiently modeled as a 56 node block graph, thereby keeping the problem size under control.

The block level partitioning has the following disadvantages.

- The RTL synthesis model of the block graph serializes the block execution. Exactly one block is active at any time. Due to this, operators in different blocks cannot be parallelized even when adequate resources are available. Thus, as the number of blocks in the design increases, the lower bound on the achievable latency tends to increase. To avoid this operators that may be executed in parallel must be in the same block. A poorly crafted block graph is bound to generate a low quality design.

- The DFG specification of a design is straightforward. Given a set of operators such as 2 input adders, subtracters and multipliers, the user expresses the design as a set of assignment statements. The extraction of the DFG from such a specification is trivial and is well understood by an average user.

  The block graph model is extracted from a high level specification in VHDL. Since the VHDL specification subset is rich (allows loops, conditionals, waits and regular signal assignment), there are several different ways in which a design can be specified. The translation from VHDL to block graph, or equivalently, the extraction of block graph is highly dependent on the specification style. For instance, a port read or a write in the VHDL specification forces a block separation. The different cases of a conditional are in separate blocks. These rules about the translation process must be well understood by the designer.

  More importantly, the block graph specification has precise synthesis [49] semantics that are strictly honored by the Asserta HLS system [50]. The performance and area of the synthesized design are closely related to the exact nature of the BBIF. Also, the amount of data communicated between adjacent blocks,[7] or equivalently, the number of edges in the block graph is dependent on the specification.

  In essence, it is possible for a naive user of the system to specify an unsuitable behavioral specification. In order to efficiently use the partitioning and synthesis environment (Section 4.5.1), the user must have a good understanding of the following 1) behavioral specification to block graph translation process 2) synthesis semantics of the block graph 3) partitioning semantics.

[7]This is determined based on data dependency analysis. In the BBIF representation [49], *all* variables that are live across adjacent blocks are passed from the source block to the adjacent destination block. This translates to an edge in the block graph.

4.6 Conclusions

In this chapter we presented novel partitioning and synthesis methodologies for DFG and block graph partitioning for RCs with multiple-FPGA and a shared memory. The unique feature of the partitioners is that they seamlessly transform inter-FPGA nets into memory based communication thereby alleviating the pin-out and interconnection bottleneck. This is done at the expense of minimal increase in the latency of the design. Results of block graph partitioning illustrates the effectiveness of dynamic design space exploration during partitioning.

Dynamic design space exploration was possible due to coarser-grain specification models. In the case of DFG partitioning, the problem size is dominated by the different possible partition configurations. However, in the case of block graph partitioning, the problem complexity shifted to design space exploration. Results show that coarser grain partitioning with effective dynamic design space exploration improves performance and run-times for large designs. In order to efficiently utilize an RC, the focus of the research must be on *specification*, *implementation*, *cost-evaluation* models, and *design space exploration* techniques. Research efforts to develop new partitioning heuristics, or improve existing algorithms, are useful, but should not be the primary focus in the context of RCs.

The efficiency of block level partitioning is dependent on the quality of the input specification. In general VLSI CAD frameworks address complex problems that make it almost impossible to find optimal solutions without the efficient involvement of the user. Most state-of-the-art VLSI CAD tools are developed for use by qualified VLSI designers. Given this, it is appropriate to assume a fair amount for user contribution to make such CAD frameworks successful.

The inherent drawback of the block graph model is its single thread of control. Exactly one block is active at any time. Since blocks cannot be fragmented across segment boundaries, at most one FPGA device is active at any time. This is clearly not an efficient model for RCs with several FPGAs. To avoid such under utilization of resources multi-threaded models for synthesis and partitioning must be considered. Efficient multi-threaded models for RCs with distributed memories are addressed as part of the SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) system [26, 68, 78, 77] in the next chapter.

Table 4.2: **Results for DFG Partitioning**

Example	RC Architecture (Name, FPGA -Type)	Segment Areas (CLBs)	Total Area (CLBs)	Latency (c-steps)	GA Run-Time (h.m.s)
Vprod	2-FPGAs, xc4005	139, 119	258	64	1m
StatFn	2-FPGAs, xc4005	141, 93	234	69	1m40s
Reverb	Wildforce, xc4013	356, 516, 292, 292	1456	83	3m35s
FIR	Wildforce, xc4013	210, 0, 504, 330	1044	117	2m30s
Ellip	Wildforce, xc4013	452, 0, 564, 452	1468	126	7m15s
FFT1D	Wildforce, xc4013	0, 218, 518, 0	736	91	3m10s
FFT2D	Wildforce, xc4013	128, 544, 461, 574	1707	187	11m5s
MatMult	2-FPGAs, xc4025	896, 896	1792	125	6m43s
DCT4x4	Wildforce, xc4085	2412, 1686, 1149, 2224	7471	597	1h15m
DCT4x4	WildChild, xc4036	1080, 1210, 1267, 1101, 1181, 1233, 840, 726	8638	1084	4h
DCT4x4	Wildfire, xc4025	952, 184, 710, 543, 212, 660, 691, 778, 811, 836, 790, 605, 713, 590, 672, 884	10631	1295	6h
DCT8x8	Wildfire, xc4085	1786, 2239, 2611, 1990, 2046, 1907, 2370, 2446, 2412, 1958, 2446, 2691, 2867, 2201, 2577 2009	36556	14045	142h

Table 4.3: **Results of Layout Synthesis and On-board Testing**

Example		Area (CLBs)		Latency (c-steps)	
		Estimated	Actual	Estimated	Actual
Vprod	s_1	139	158	64	65
	s_2	119	135		
StatFn	s_1	141	165	69	72
	s_2	93	102		
MatMult	s_1	816	729	125	123
	s_2	816	728		

Table 4.4: **Design Data for DFG Partitioning**

Example	Num Blk	Num Edg	Num Loop	$\mathcal{L}_{min}$ c-stp	$\mathcal{L}_{max}$ c-stp	$\mathcal{A}_{max}$ CLBs	$\mathcal{A}_{min}$ CLBs
Find	22	30	3	584	584	746	746
ALU	9	12	0	18	21	151	149
MeanVar	11	10	0	37	56	330	196
Reverb	10	10	0	38	45	910	639
FIR	10	9	0	85	93	643	504
Ellip	21	21	0	59	70	1006	831
FFT1D	16	20	1	79	93	297	266
FFT2D	32	40	2	150	194	1500	905
MatMult	26	34	2	160	232	1349	978
DCT4x4	104	136	8	357	517	8129	5541
DCT8x8	56	72	2	517	1573	72008	19874

Table 4.5: **Results for Block Graph Partitioning**

Example	RC Architecture (Name, FPGA -Type)	Segment Areas (CLBs)	Total Area (CLBs)	Latency (c-steps)	SA Run-Time (h.m.s)
Find	2-FPGAs, xc4013	535, 481	1016	596	4s
ALU	2-FPGAs, xc4003	52, 91	143	34	6s
MeanVar	2-FPGAs, xc4005	104, 159	263	51	8s
Reverb	Wildforce, xc4013	439, 451, 0, 0	890	52	4s
FIR	Wildforce, xc4013	0, 301, 0, 464	765	97	10s
Ellip	Wildforce, xc4013	436, 132, 365 0	933	80	9s
FFT1D	Wildforce, xc4005	0, 169, 166, 0	335	109	8s
FFT2D	Wildforce, xc4013	528, 314, 405, 0	1247	179	24s
MatMult	2-FPGAs, xc4025	669, 798	1467	160	20s
DCT4x4	Wildforce, xc4085	1263, 2432, 2462, 1988	8145	556	5m15s
DCT8x8	WildChild, xc4036	2250, 2478, 2907, 3051, 2572, 2797, 2508, 3026	21589	1752	1h40m
DCT8x8	WildChild, xc4036	2545, 2900, 2611, 2763, 2858, 2797, 2878, 2831	22183	1806	2h05m

Chapter 5

Partitioning with Synthesis

5.1 Introduction

The Reconfigurable Computer (RC) consisting of multiple FPGA devices, memory banks, and device interconnections, offers a variety of resources but is limited in hardware. Design automation for RCs from a behavioral specification consists of three fundamental problems: (i) Temporal Partitioning – This generates a sequence of temporal segments, each of which utilizes *all* the RC resources. The temporal segments may then be executed on the RC in the specified sequence, thereby sharing all the RC resources over time; (ii) Spatial Partitioning – Each temporal segment can further be divided into spatial partitions in order to effectively utilize the multiplicity of resources available on the RC; (iii) High-Level Synthesis (HLS) – This involves synthesizing each spatial partition into an Register-Transfer Level (RTL) design intended for a device on the RC.

Figure 5.1 shows an overview of the SPARCS system [26, 68] consisting of a synthesis framework that interacts with a partitioning environment. The RC is viewed as a co-processor that is controlled by a host computer. The SPARCS system accepts a behavioral specification in the form of a Unified Specification Model (USM) [25]. The USM can capture a parallel-process specification in VHDL [27], and embodies features that are highly suited for RC synthesis. The USM is essentially a graph consisting of: (i) *task* nodes that are used to capture elements of computation in the behavior. Each task is a Control Data Flow Graph (CDFG, [10]) representing a single thread of control (a VHDL process); (ii) *logical memory* nodes that are elements of data communication between the tasks; and (iii) *flag* edges that are used to synchronize the execution of tasks. A flag between two tasks (t_i, t_j) specifies the *dependency* of t_j on t_i.

Temporal partitioning in SPARCS uses the inter-task dependencies to derive a temporal schedule of tasks. The schedule consists of a sequence of temporal segments where each segment is a subgraph of the USM. The primary goal of temporal partitioning is to minimize the *delay* of the temporal schedule, defined as $N * R + \sum_{i=1}^{N} L_i * CP$. Here, N is the number of temporal segments, R is the reconfiguration time of the RC, and CP is the user-given clock period for the design. The temporal schedule so generated has a corresponding latency constraint (L_i) on each temporal segment i. The temporal partitioner also ensures that: (i) the collection of tasks in each temporal segment after synthesis will fit within the RC, and (ii) the memory requirements for each temporal segment are within the available physical memory on the RC.

Spatial partitioning in SPARCS involves partitioning each temporal segment such that: (i) the set

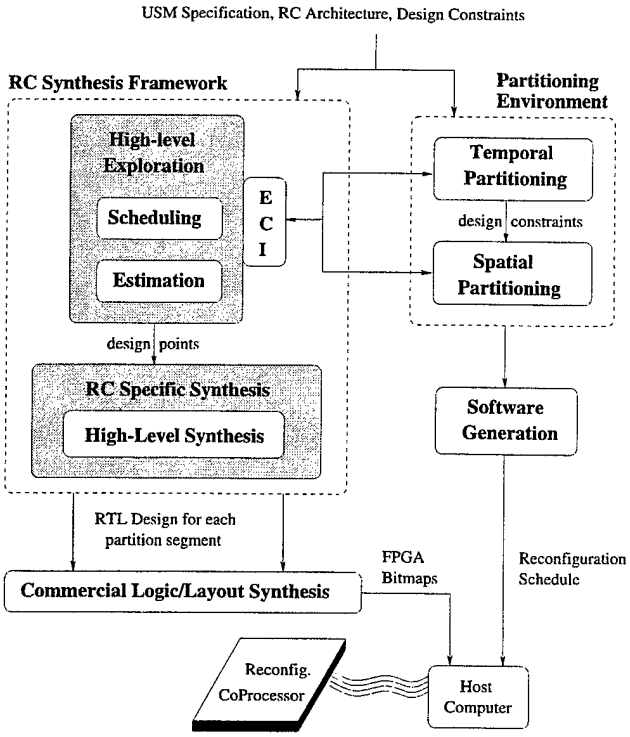


Figure 5.1: **SPARCS Design Automation System for RCs**

of tasks in each spatial partition after synthesis will fit within the corresponding device; (ii) the latency constraint is satisfied; (iii) the logical memories are mapped to the physical memory banks on the RC; and (iv) the flags and the memory buses are routed through the interconnection network on the RC. It is imperative that spatial partitioning that follows temporal partitioning be done with utmost care so as to satisfy the RC and design constraints. Henceforth, we will use the term *partition* to denote a spatial partition.

Both temporal and spatial partitioning require design estimates that are used to evaluate the partitioning costs. In order to generate efficient Register-Transfer Level (RTL) designs that implement the given behavior, HLS and partitioning techniques need to carefully select implementations that best satisfy the constraints. The synthesis framework in SPARCS allows tight integration of the partitioning environment with a design space exploration engine through an Exploration Control Interface (ECI). The ECI consists of exploration/estimation methods that a partitioning tool may call to select *design points* (possible implementations) and obtain estimates. After the partitioning and exploration is completed, the back-end HLS tool is constrained by the selected *design points* to synthesize RTL designs that satisfy these estimates.

Traditional approaches [83, 19, 48] to integrate HLS and spatial partitioning perform exploration and estimation along with partitioning. In the *traditional heterogeneous model* of HLS and spatial partitioning, the partitioner invokes a HLS estimator to obtain the area/latency of each spatial partition. Several heterogeneous systems, such as *SpecSyn* [9], *Chop* [34] and *Vulcan I* [59], focussed on providing good design estimates while not performing complete HLS. Later, researchers (COBRA-ABS [1], *Multipar* [86]) developed a completely *homogeneous model*, wherein HLS and partitioning are performed in a single step. The COBRA-ABS system has a Simulated Annealing (SA) based model and *Multipar* has an ILP based model for synthesis and partitioning.

However, unification of spatial partitioning and HLS into a homogeneous model adds to the already complex sub-problems of HLS, leading to a large multi-dimensional design space. Therefore, the cost (design automation time) of having a homogeneous model is very high, i.e, either the run times are quite high (COBRA-ABS [1]) or the model cannot handle large problem sizes (*Multipar* [86]). The traditional heterogeneous model, although less complex, also has a significant drawback of performing exploration on a particular partition segment, *which is only a locality of the entire design space.*

In this chapter, we propose a *spatial partitioning knowledgeable exploration technique* that combines the best flavors of both the models. The exploration technique has the capability to simultaneously explore the design space of multiple spatial partitions. This enables exploration and spatial partitioning to generate constraint satisfying designs in cases where the traditional heterogeneous model fails. In [69], we introduced the idea of a partitioning-based exploration model for single-threaded behavioral specifications. In this chapter, we extend this to parallel-process (USM) specifications and present the integration of design space exploration with spatial and temporal partitioning in the SPARCS [26] system.

The rest of the chapter is organized as follows. Section 5.2 describes the proposed partitioning knowledgeable exploration model for a USM specification. Section 5.3 presents the exploration algorithm in detail with an illustrative example. Section 5.4 presents the integration with the temporal and spatial partitioning in SPARCS. Section 5.5 presents results comparing the traditional and proposed exploration techniques. Finally, we present a summary in Section 5.6.

5.2 Partitioning Knowledgeable Exploration Model for the USM

The USM embodies a *task graph* that consists of a collection *tasks* (N_{tasks}) and edges representing dependencies (flags) between them. Each task is a CDFG consisting of *blocks* of computation and edges representing control flow. Each block in a task in-turn has a simple data flow graph, while the collection of blocks (in a task) represent a single-thread of control. The collection of tasks the USM represent a parallel control-thread model.

Definitions for Partitioned Task Graph: We define following terms with respect to our partitioned task graph model:
• A *partition* $P_i \subseteq N_{tasks}$, is a subset of tasks in the task graph.
• A *configuration* $C_{set} = \{P_i \mid (P_i \cap P_j = \emptyset) \wedge (T_i \in N_{tasks} \Rightarrow \exists P_k : T_i \in P_k)\}$ is a set of mutually exclusive partitions of all the tasks.
• A *design point* $DP_{i,k}$ corresponds to a specific implementation i of a task k. A design point is essentially a collection of *schedules* [10], one for each block in the CDFG of the task.
• A $L(t)$ is the latency of the task t, defined as the number of clocks cycles per input vector.
• $L_{min}(t)$ is the fastest latency of the task t, corresponding to the ASAP schedules of all its blocks.
• $L_{max}(t)$ is the slowest latency of the task t, corresponding to the slowest (smallest resource bag) schedules of all its blocks.
• $A_{min}(t)$ *and* $A_{max}(t)$ represent the smallest and largest design areas of task t corresponding to the slowest and fastest schedules, respectively.
• A *design space* of a task t is the set of all possible design points bounded by $L_{min}(t)$ and $L_{max}(t)$. Further, the design space of a partition is the union of the design spaces of all tasks in that partition.

For the partitioned USM shown in Figure 5.2(a), $C_{set} = \{P_1,\ P_2\}$, where $P_1 = \{T_1, T_2, T_3\}$ and

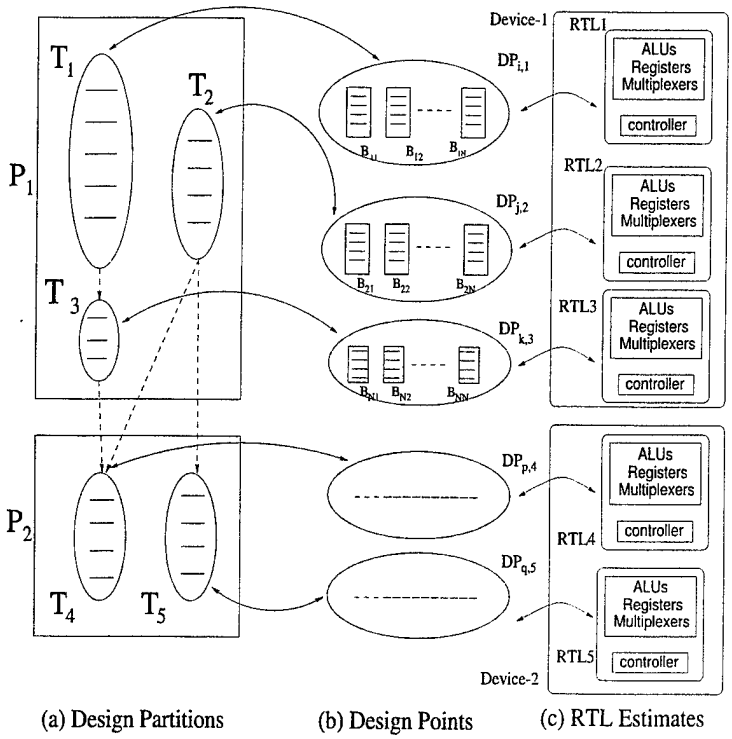


Figure 5.2: **The USM exploration model**

$P_2 = \{T_4, T_5\}$. Figure 5.2(b) shows the design points corresponding to each task. Note that within each design point, a collection of block-level schedules are maintained. From each design point, an RTL design for the corresponding task can be synthesized. In addition, the RTL resource requirements for each individual block of any task is also maintained. Note that the blocks belonging to a task share all the datapath resources and a single finite state machine controller. Thus, for each design point detailed RTL design estimates are maintained. As shown in Figure 5.2(c), each partition P_i is synthesized as a collection of RTL designs for the corresponding device in the RC.

The exploration model currently does not share hardware between tasks, instead, performs an efficient allocation of the device area to the tasks that are assigned to that partition. In addition, the exploration model attempts to minimize design latency by exploiting the task-level and operation-level parallelism. Nevertheless, the model can be changed to allow sharing by simply modifying the RTL estimation mechanism and introducing a suitable controller model [32].

Design Constraints: The goal of the exploration process is to generate design points for any given USM configuration, such that the following design constraints are best satisfied:
• *Design Latency*($L_{constraint}$): is a constraint on the set of tasks belonging to one temporal segment. It is defined as:
$\sum_{t \in CP} L(t) \leq L_{constraint}$, where $CP \subseteq N_{tasks}$, is the *critical path* of tasks in the graph. We define the critical path as the path that determines the largest total latency.
• *Device Area*($DeviceArea_k$): The target architecture consists of multiple devices each of which can have different area. Therefore, each device k imposes an area constraint on the corresponding partition P_k of the USM, defined as: $DesignArea(P_k) \leq DeviceArea_k$, where $DesignArea(P_k)$ is the estimated RTL design area of partition P_k.

• *Component Library*: is an user-specified RTL component library from which the exploration engine selects a set of resources (ALUs) to perform scheduling and allocation. The user may specify a specific component library for each task.

5.2.1 The Exploration Control Interface

The exploration technique provides an Exploration Control Interface (ECI) that facilitates tight integration with any partitioning algorithm. The interface consists of a collection of exploration and design area estimation methods that generate design points and area estimates for the current USM configuration. *These methods can be collectively used to control the trade-off between the time spent in exploration and the amount of design space explored.* Here we provide a brief description of the intent of these methods. In the following section, we will correlate these methods to the exploration algorithm. These methods are listed in the decreasing order of their complexity or the amount of design space explored, Therefore, they also fall in the decreasing order of the time spent in exploration.

Explore Design(P_{set}): Given a current configuration (C_{set}) and a subset of partitions ($P_{set} \subseteq C_{set}$), the exploration engine attempts to generate a design point for each partition in P_{set} such that the design latency constraint and all device areas constraints are best satisfied. For example in Figure 5.2, we can re-generate new design points for the five tasks, by simultaneously exploring both partitions P_1 and P_2.

Explore Partition(P_k): This is a more constrained exploration that generates new design points for the tasks in one partition P_k. The goal again is to satisfy the design latency and the device area constraints. Note that this method does not change the design points of tasks in any partition other than P_k. For example in Figure 5.2, we can just generate new design points for tasks T_4 and T_5 by exploring only partition P_2.

Explore Task(T_i): This method reschedules only the task T_i at various time constraints, until the latency and area constraints are met or all possible schedule lengths for the task T_i have been explored.

Estimate Design(P_k): For partition P_k, at any time during the design process, there exists a collection of design points. Using these design points, the method determines RTL design parameters for each task and estimates the design area of partition P_k.

5.3 The Exploration Algorithm

In this section, we describe the exploration algorithm elaborately. However, a reader may proceed to the following section without any loss in understanding the overall exploration and partitioning methodology presented in this chapter.

The exploration algorithm is shown in Figure 5.3. Given a subset of partitions $P_{set} \subseteq C_{set}$, the algorithm determines the set of tasks $T_{set} = \cup_{P_k \in P_{set}} P_k$ that need to be explored. *The goal of the algorithm is to generate design points for the tasks in T_{set} such that the design constraints are best satisfied.* For each task $t \in T_{set}$, the algorithm initially generates a design point $DP_{fast,t}$ corresponding to the fastest schedule. Therefore, initially each task would have a worst case area but least latency ($L_{min}(t)$). The design points (or schedules) for the rest of the tasks $N_{tasks} - T_{set}$ are left untouched.

Algorithm: USM_Exploration ▷ Explore_Design(P_{set})
Input: Design constraints and a set of task partitions P_{set}
Output: A DP_i, $\forall T_i \in P_i$, $\forall P_i \in P_{set}$
and the RTL estimates $\forall P_i \in P_{set}$

Begin
1 $T_{set} = \{T_i \mid T_i \in P_i,\ \forall P_i \in P_{set}\}$
2 $\forall T_i \in T_{set}$: Initialize(T_i)
3 Iteration = 0 ▷ *relax-tighten loop*
4 **while** (Iteration < UpperBound)
5 Compute *critical path*
6 Compute *design costs*
7 Update S_{best}
8 **if** (design fits) **then exit**
9 T_{list} = sorted T_{set}, using cost functions:
10 1. Decreasing PAV_t
11 2. Non Critical tasks followed by Critical tasks (C_t)
12 3. Increasing LA_t cost
13 T_r = Select Task from T_{list} for Relaxation
14 T_t = Select Task from T_{list} for Tightening
15 **if** ($T_r \neq \emptyset$) **then**
16 *Relax*(T_r) ▷ *bock-level exploration*
17 **else if** ($T_t \neq \emptyset$) **then**
18 *Tighten*(T_t) ▷ *block-level exploration*
19 **else exit**
20 **end if**
21 **end while**
22 **if** (design did not fit)
23 Restore using S_{best}
24 Re-compute *critical path and design costs*
25 **end if**
End

Figure 5.3: **USM Exploration Algorithm**

The algorithm performs exploration in a loop (lines 4-21), where each iteration *relaxes* or *tightens* the schedule of a task. Relaxing a task corresponds to a latency increase and an area reduction, and tightening works vice versa. During each iteration, the *critical path* and the *design costs* are evaluated. The algorithm maintains the *best solution* (S_{best}, a collection of design points $\forall\ t\ \in\ T_{set}$) obtained so far, defined as the one that has the least *total* $AreaPenalty = \sum_{P_k \in C_{set}} AO(P_k)$, where the *Area Overshoot* $AO(P_k) = \begin{cases} \Delta A_k & \text{if } \Delta A_k > 0 \\ 0 & \text{otherwise} \end{cases}$, and $\Delta A_k = DesignArea(P_k) - DeviceArea_k$.

At the core of the exploration algorithm is a collection of cost functions that determine the task to be selected for relaxation (T_r) or tightening (T_t). Using these cost functions the tasks in T_{set} are sorted to form a priority list (P_{list}). While selecting a task for relaxation, the priority list is traversed from left to right, and for tightening from right to left. Each cost function captures an essential aspect of the partitioning-based exploration model and these functions collectively guide the exploration engine in finding a constraint satisfying design. These cost functions have been

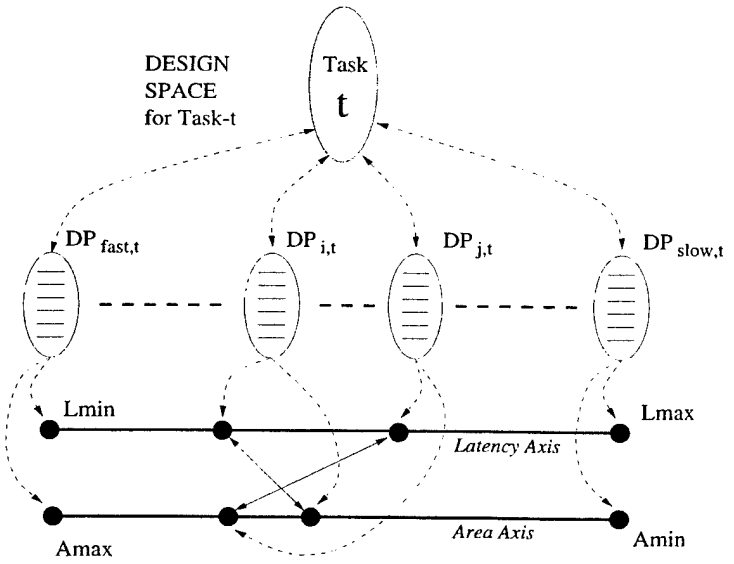


Figure 5.4: **Design Space of a Task**

listed in the order in which they are applied for sorting the list of tasks:

- *Partition Area Violation* (PAV_t): represents the area violation of the partition to which the task belongs. $PAV_t = \frac{DesignArea(P_k) - DeviceArea_k}{DeviceArea_k}$, *and* $t \in P_k$. The tasks are ordered in increasing PAV_t such that, tasks belonging to the most occupied device are selected for relaxation and tasks belonging to the least occupied device are selected for tightening. Note that all tasks belonging to the same partition will have the same PAV.

- *Criticality* (C_t): A critical task C_t is one that is on the critical path. Between the set of tasks that belong to one partition, those that do not fall on the critical path are ordered before those that are on the critical path. This is because, non-critical tasks are good candidates for relaxation, since the design latency will not be increased. Similarly, critical tasks are good candidates for tightening, since the design latency will be decreased.

- *Latency-Area Tradeoff* (LA_t): This is the most important cost function that determines the task that is selected among those have equal PAV_t and C_t. For a task t, and the corresponding design point $DP_{i,t}$, we define the latency-area tradeoff cost as follows:

$$LA_t = \mathcal{L}_{norm}(DP_{i,t}) + \mathcal{A}_{norm}(DP_{i,t}), where \tag{5.1}$$

$$\mathcal{L}_{norm}(DP_{i,t}) = (\frac{\mathcal{L}(DP_{i,t}) - L_{min}(t)}{L_{max}(t) - L_{min}(t)}) * 100, and$$

$$\mathcal{A}_{norm}(DP_{i,t}) = (\frac{A_{max}(t) - \mathcal{A}(DP_{i,t})}{A_{max}(t) - A_{min}(t)}) * 100$$

We will explain the terms used in this cost:

  - $\mathcal{L}_{norm}(DP_{i,t})$ represents the normalized latency percentage of the current design point with respect to the latency bounds.
  - $\mathcal{A}_{norm}(DP_{i,t})$ represents the normalized area percentage of the current design point with respect to the latency bounds.

- $\mathcal{L}(DP_{i,t})$ represents the current latency of the task t, with respect to the current design point $DP_{i,t}$.
- $\mathcal{A}(DP_{i,t})$ represents the current area of the task t, with respect to the current design point $DP_{i,t}$.

We will explain this cost function using the pictorial view of the design space of a task shown in Figure 5.4. For a task t, the set of all design points can be ordered in the latency axis from its $L_{min}(t)$ to $L_{max}(t)$. Correspondingly, the design points for the task t can be ordered on the area axis from $A_{max}(t)$ to $A_{min}(t)$. As shown in Figure 5.4, for any two design points $DP_{i,t}$ and $DP_{j,t}$ their ordering in both the latency and area axis need not be the same. However, the issue of concern is *how close or far a design point is from the respective bounds*. The cost $\mathcal{L}_{norm}(DP_{i,t})$ is a metric for measuring the distance of the design point $DP_{i,t}$ from the *latency lower bound* $L_{min}(t)$. Similarly, the cost $\mathcal{A}_{norm}(DP_{i,t})$ is a metric for measuring the distance of the design point $DP_{i,t}$ from the *area upper bound* $A_{max}(t)$. Both the costs have been *normalized* within their ranges such that they can be summed up to provide a *closeness factor* (LA_t) of the design point with respect to the latency and area lower bounds.

A low value LA_t implies that the tasks' current area is close to its upper bound and the current latency is close to its lower bound. This means that tasks with low LA_t are good choices for relaxation so that their latency can be increased and their area can be reduced. Similarly, tasks with high LA_t are good choices for tightening. The tasks in priority list are ordered in increasing values of LA_t.

After these costs are applied and the priority list is ordered, the algorithm selects a task for relaxation or tightening. If there exists a task whose latency can be relaxed and still remains within the bound, then the algorithm relaxes it, otherwise a task is selected and tightened. In order to relax or tighten a task, the algorithm invokes the block-level exploration algorithm [69]. The block-level algorithm, based on the internal cost metrics of the task, selects and re-schedules the *best block* [69] within that task. For scheduling a task, we use a low-complexity time-constrained scheduling algorithm [67]. The criteria for the relaxation and tightening a task are: (i) the tasks' latency should remain within the bounds, and (ii) the design latency should remain within the given constraint ($\sum_{T_i \in CP} Latency(T_i) \leq L_{constraint}$).

The relax-tighten loop stops when any one of these conditions are met: (i) the design fits – all device area constraints are satisfied, (ii) none of the tasks can be relaxed or tightened, or (iii) A lot of exploration time (iterations) has been spent. This is provided so that the exploration time can be cut-off for large design spaces. At the end of the relax-tighten loop, if the design did not fit the best solution is restored.

5.3.1 Implementing the ECI

Here, we provide a short description of the algorithms used by the ECI methods and their correlation to the exploration algorithm.

Explore Design(P_{set}): This method is implemented by invoking the exploration algorithm on the given P_{set}. The method generates a schedule for each task in $P_{set} = \{T_i \mid \forall P_k \in P_{set}, T_i \in P_k\}$, and estimates the design areas of all partitions in P_{set}, such that the constraints are best satisfied. *Since the exploration engine maintains the current configuration of the entire USM, it can find constraint satisfying solutions where a traditional exploration/estimation technique would fail.*

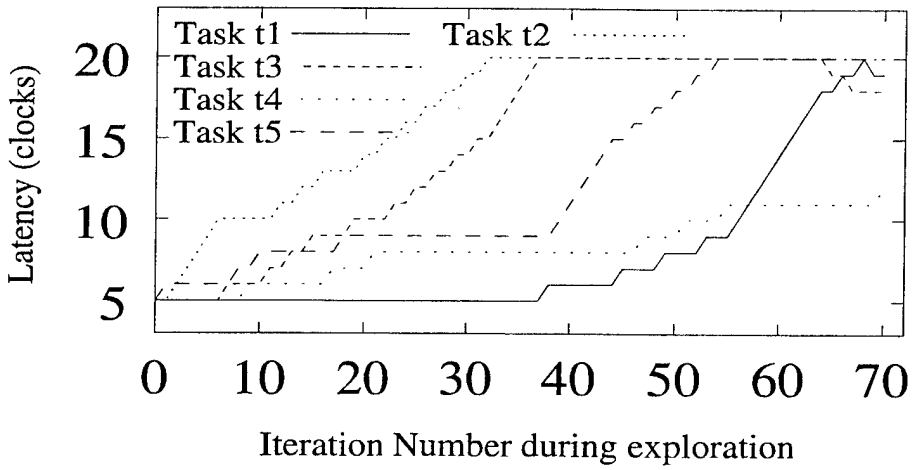


Figure 5.5: **Task Latencies During Exploration**

Explore Partition(P_k): This method is also implemented by invoking the exploration algorithm on $P_{set} = \{P_k\}$. This method is equivalent the *traditional* exploration technique that only performs a *local search* of the design space of one partition. Hence, this is a more constrained method that may not be able to satisfy the design latency and device area constraints as well as the *Explore_Design()* method.

Explore Task(t): This method invokes the block-level exploration algorithm [69] on all blocks in task t. The goal of the algorithm to generate a set of schedules for all blocks in the task t such that the area constraint on partition $P_k \ni t$ and design latency are best satisfied.

Estimate Design(P_k): When this method is invoked there always exist a design point for each task $t \in P_k$. This method performs a post-scheduling estimation to estimate RTL design area of each task. Currently, the partition area is computed as the sum of all task areas. If a shared datapath is implemented, then the estimation method need may be modified accordingly.

5.3.2 Illustrative Example

We will use the example shown in Figure 5.2(a) to illustrate the effectiveness of the task-level exploration using the *Explore_Design()* method. The example has five tasks $T_1 - T_5$ in two partitions, $P_1 = \{T_1, T_2, T_3\}$ and $P_2 = \{T_4, T_5\}$. The behavior of each task has four vector products consisting of sixteen 8-bit multiplications, eight 16-bit additions and four 17-bit additions. The design space for each task on the latency axis varies from 5 to 20 clocks and on the area axis from 120 to 5,200 CLB s. It can be seen that the design space of the entire design is very large, $((20-5)^5)$ possible latency combinations for the five tasks.

We have shown as a collection of three plots, the progress of the exploration algorithm. The plot in Figure 5.5 shows the latency of each task during each iteration of the algorithm. The plots in Figure 5.6 show the variation of design area and latency over iterations of the algorithm. A latency constraint of 49 clocks was provided as a constraint for the exploration algorithm. Area constraints of 900 clbs and 570 clbs were imposed on partitions P_1 and P_2, respectively.

Initially, all tasks have their fastest (latency = 15) implementation and their design areas in both partitions are at the upper bounds. Initially T_5 that is not on the critical path is relaxed leading

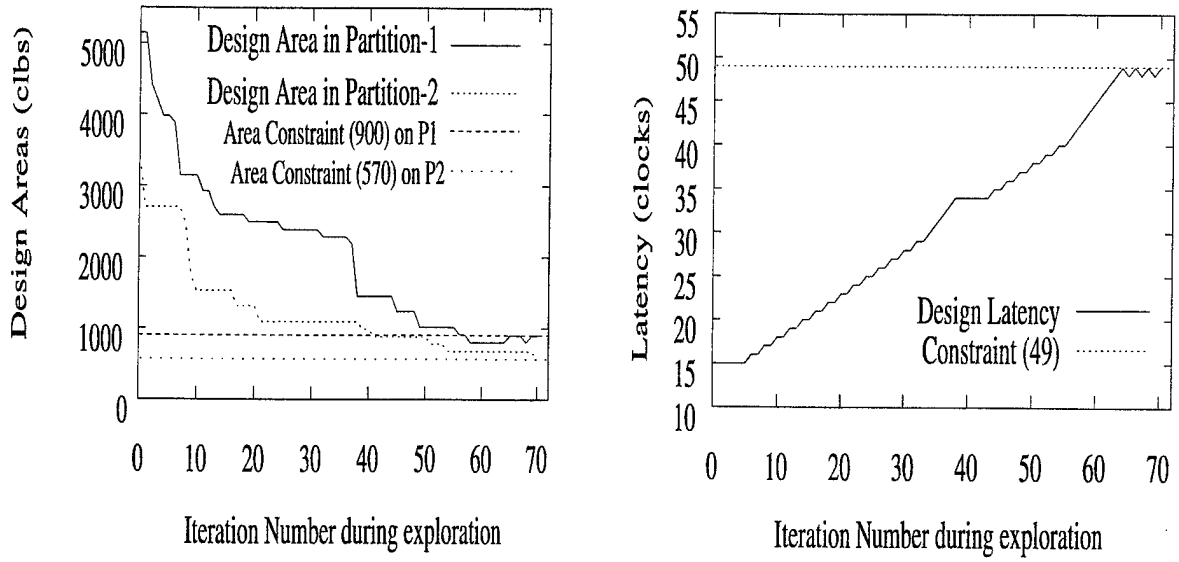


Figure 5.6: **During Exploration, Iteration Vs. : (a) Design Area and (b) Design Latency**

to a drop in the area of P_2. Now, the area violation in P_1 is much higher hence the algorithm proceeds to relax the tasks $\{T_1, T_2, T_3\} \in P_1$. Task T_2 being non-critical is selected and relaxed for the next 6 iterations leading to a drop in the area of P_1. Again the area violation of P_2 is higher, hence T_5 (non-critical) is relaxed leading to a significant drop in the area of P_2. During the next 28 iterations (10-38), the algorithm attempts to relax the tasks in P_1 which now has a high PAV. The tasks T_2 and T_3 are relaxed alternatively, and at iteration 38, the area of P_1 has decreased considerably. Hence, T_5 (non-critical) is selected and relaxed for the next 10 iterations with a few relaxations of T_4 which is critical. At iteration 56, finally the area of P_1 falls below the constraint. From this point, the algorithm attempts further combinations to fit P_2. It can been seen that when the design latency reaches the constraint value (at iteration 61), the algorithm does not allow any further relaxation, thereby keeping the design latency always within the constraint. During iterations iteration 61-67, $T_3 \in P_1$ is tightened and the slack obtained is used to finally relax T_4 to fit P_2.

This way the algorithm effectively distributes the latency among the tasks, so as to pick design points that effectively satisfy the individual device area constraints.

5.4 Integrating Exploration and Partitioning in SPARCS

The SPARCS design flow consists of temporal partitioning followed by spatial partitioning and finally high-level synthesis, as described earlier in Section 5.1. In this section, we will describe the integration of design space exploration with temporal and spatial partitioning algorithms in SPARCS.

5.4.1 Interaction with Temporal Partitioning

For a temporal partitioner, there are two ways to integrate the exploration engine. One approach is through a spatial partitioner that in turn interacts with the exploration engine. The ECI

```
Algorithm: Template for GA or SA
1  C_current = Random Initial Configuration
2  ExpEngine.Initialize(C_current)
3  Gen/Temp = Initial Generation or Temperature
4  while (Gen/Temp  <  Final Value)   ▷ GA or SA loop
5    while (PopSize/Iters  <  Max Value)   ▷ GA(PopSize) or SA(Iters) loop
6      C_new = Perturb(C_current)
7      for each (Temporal Segment G_i ∈ USM)
8        ExpEngine.SetConfig(G_i, C_new)
9        ExpEngine.SetLatency(L_i)
10         ExpEngine.Task_Level_Exploration   ▷ single or multi-partition
11         ExpEngine.Estimate_Design()
12     end for
13     if (acceptable(C_new,C_current,Temp))
14       C_current = C_new
15     end if
16   end while   ▷ PopSize/Iter loop
17 end while   ▷ Gen/Temp loop
```

Figure 5.7: **Template of a GA-based or SA-based USM Partitioner**

directly supports a tight interaction with a spatial partitioner. Therefore, it would be ideal for a temporal partitioner to interact through a spatial partitioner. This way it will have accurate estimates on the utilization of RC resources. However, this approach would be time consuming since spatial partitioning is done in conjunction with temporal partitioning.

For temporal partitioners that are based on optimal models such as ILP, such as in SPARCS [36], this approach will be impractical. A second approach to integrate the exploration engine with a temporal partitioner, is to assume that the RC has a single device whose area is the sum of the areas of the all the devices. In SPARCS, the temporal partitioner assumes such a lumped model of the RC and without considering the effects of spatial partitioning in detail. Nevertheless, SPARCS incorporates a detailed feedback to temporal partitioning, in the case when spatial partitioning fails on any temporal segment.

The lumped RC area is set as a constraint to the exploration engine and all tasks are placed in one partition. The *Explore_design()* method is invoked several times with varying design latency constraints at equidistant points within the bounds on the *entire design latency* (for all tasks). For each invocation of *Explore_design()*, a design point is generated for each task. Thus, prior to temporal partitioning in SPARCS, several design points are generated for each task in the USM. This enables the temporal partitioner to contemplate multiple implementations of a task, while trying to optimize design latency. At the end of temporal partitioning, a latency constraint is imposed on each temporal segment such that the entire *design delay* (see Section 5.1) is minimized.

5.4.2 Interaction with Spatial Partitioning

Following temporal partitioning, each temporal segment is spatially partitioned by the SPARCS system. As described earlier in Section 5.1, the goals of the spatial partitioning are to fit the temporal segment on the devices while satisfying the latency, memory and interconnection constraints.

The spatial partitioning system [26, 81] consists of two partitioning algorithms: (i) A Simulated Annealing (SA-based) and (ii) Genetic Algorithm (GA-based). These algorithms interact with the exploration engine in order to *dynamically* generate design points that satisfy the device area and latency constraint on each temporal segment. The other two constraints on memories and interconnections are handled within the spatial partitioner.

Figure 5.7 presents an abstract template representing both algorithms. The boxed lines indicated places of interaction between the partitioner and the exploration engine. Initially, one (or more) random configuration(s) are generated and the exploration engine is initialized with these. The loop in line-4 represents the outer-loop of the GA *generations* or the SA *temperatures*. The loop in line-5 represents the inner-loop of the GA *population size* or the SA *iterations per temperatures*.

In order to achieve efficient memory utilization, *the spatial partitioning of all temporal segments are performed simultaneously* by the USM spatial partitioner [26, 81]. For this purpose, these algorithms maintain a *super-configuration* that is the set of all configurations (spatial partitions) over all temporal segments. During each iteration, the algorithms generate a new super-configuration (C_{new}) by perturbing the existing super-configuration ($C_{current}$). Note that, any super-configuration is composed of a set of configurations, one corresponding to each temporal segment (subgraph $G_i \in USM$) of the USM. Here, G_i contains the set of all tasks belonging to the temporal segment i.

The lines 8-11 pass each such configuration (of G_i) to the exploration engine and set the corresponding latency constraint (L_i) of the temporal segment. Then, the exploration algorithm is invoked on a temporal segment and the design areas of all spatial partitions in that temporal segment are estimated. This way, exploration and estimation are performed for all temporal segments ($G_i \in USM$) in a sequence. During the spatial partitioning process, when any configuration is accepted (at line-14) by the GA or SA, the exploration engine is again invoked (the same way) to generate new design points and estimates design areas.

Note: For experimentation, we developed two versions for each partitioning algorithm. The first one represents the *traditional* model of exploration where at any time during spatial partitioning (at line-10) only the single-partition exploration is performed using the *Explore_Partition()* method. The other version represents the *proposed* model where at line-10 the multi-partition exploration is always performed using the *Explore_Design()* method. Thus, the proposed model always performs a *partitioning knowledgeable exploration* on multiple spatial partitions, whereas the traditional model performs a local search on individual spatial partitions.

5.5 Results

First, we present results demonstrating the effectiveness of the multi-partition exploration technique as compared to the traditional model of exploring a single partition. Then, we present results of some designs that were synthesized through SPARCS and tested on a commercial RC

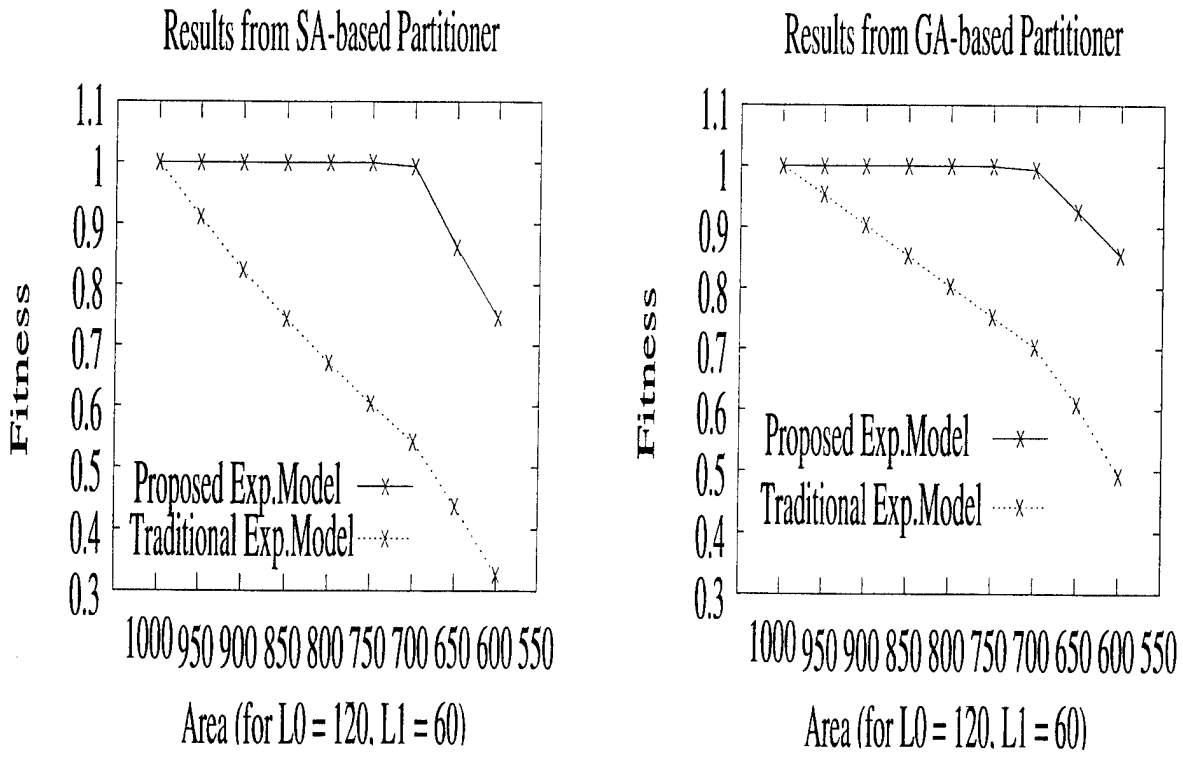


Figure 5.8: **USM Exploration and Partitioning Results for DCT**

board.

5.5.1 Exploration Results from GA-/SA-based Spatial Partitioners

We have run a number of experiments on a constructed DCT example consisting of 12 tasks. Each task has four vector products consisting of sixteen 8-bit multiplications, eight 16-bit additions and four 17-bit additions. The design space for each task on the latency axis varies from 5 to 20 clocks and on the area axis from 120 to 5,200 CLB s. It can been seen that the design space of the entire design is very large; $(20-5)^{12}$ possible design points, or latency combinations for all tasks.

We have considered the *Wildforce* [84] target architecture consisting of four FPGA devices, with fixed and programmable interconnections and 32 K memory banks. The DCT example uses two memory segments with 16 words each for the input and the output 4x4 matrices. The there are eights flags that synchronize the execution of the tasks. The twelve tasks in the DCT example were temporally partitioned into two segments with the 9 tasks in one and three in the other. We ran both the GA and SA spatial partitioners on a *Sun Workstation* with a 128MB RAM and a 143 Mhz *Sparc-5* processor. Both partitioners have a built-in Wildforce-specific router and memory estimator [81], that handle the interconnection and memory constraints.

We fixed the latency constraints of the two temporal segments at their upper bound values (120 and 60 clocks) and ran both the partitioning algorithms (SA and GA) by varying the device area constraint. Figure 5.8 shows two plots of the results from the SA-based and the GA-based partitioners. Each plot has the device area constraint on the x-axis and solution *fitness* [26, 81] on the y-axis. Fitness is a measure of the solution quality in terms of the area, memory and

Table 5.1: **USM Partitioning Results (with fitness < 1) for DCT**

A_{device}	Algo.	Exp.Model	Fitness	Design Areas in $TP0$	Design Areas in $TP1$
700	SA	Single	.541	{996, 695, 664, 664}	{0, 355, 332, 332}
		Multi	.994	{702, 499, 468, 468}	{234, 23, 0, 680}
	GA	Single	.702	{1027, 664, 664, 664}	{355, 332, 0, 332}
		Multi	.994	{702, 468, 265, 702}	{234, 0, 257, 545}
650	SA	Single	.436	{996, 695, 664, 664}	{0, 355, 332, 332}
		Multi	.862	{468, 702, 490, 585}	{234, 234, 22, 545}
	GA	Single	.608	{664, 683, 996, 686}	{332, 0, 332, 354}
		Multi	.925	{588, 702, 487, 468}	{256, 545, 234, 0}

Table 5.2: **USM Partitioning Results for FFT**

$L_{constraint}$					Design Areas	
TP0	TP1	A_{device}	Exp.Model	Fitness	TP0	TP1
154	80	576	Single	1.0	{514, 572, 558, 514}	{94, 300, 323, 0}
			Multi	1.0	{438, 558, 484, 553}	{0, 306, 402, 0}
162	82	576	Single	1.0	{514, 572, 558, 514}	{94, 300, 323, 0}
			Multi	1.0	{439, 560, 459, 481}	{113, 307, 285, 0}
158	80	570	Single	0.995	{514, 572, 558, 488}	{268, 126, 323, 0}
			Multi	0.995	{514, 572, 558, 488}	{268, 126, 323, 0}
160	80	570	Single	0.996	{462, 572, 558, 436}	{268, 126, 323, 0}
			Multi	0.996	{462, 572, 558, 436}	{268, 126, 323, 0}

interconnect constraints. Any solution always satisfies the latency constraint. A fitness value of 1 indicates that all constraints are satisfied and a *lower fitness* value indicates a *higher violation* of constraints.

Each plot, has two curves representing the solutions generated by invoking the traditional single-partition (dashed lines) and proposed multi-partition exploration models (solid lines) during spatial partitioning. As shown in both plots, for an area constraint of 1000 clbs (and higher), both exploration models of exploration found constraint satisfying solutions. As we tighten the area (lower than 1000 clbs), the SA and GA versions that perform a multi-partition exploration find constraint satisfying solutions whereas their traditional counterparts do not. In fact for the traditional models, we can see that the solution quality becomes poorer (fitness < 1) with tighter area constraints. For all these cases, the proposed model found a solution in a few seconds to minutes, whereas, the traditional model could not even after running for over 4-6 hours. This shows that the solution found by the multi-partition exploration doesn't even exist in the local search space of the traditional single-partition exploration.

At tight area constraints (650 and lower) both models do not find constraint satisfying solutions. Nevertheless, the solution generated by the proposed model is superior (higher fitness) to that of the traditional model. Table 5.1 lists the resulting design areas (in all spatial partitions, for the two temporal segments $TP0$, $TP1$) for solutions whose fitness are less than 1. We see that even in all these cases the solutions generated the proposed model is better (in area) than that of the traditional.

In Table 5.2 we have shown the results for partitioning the Fast Fourier Transform (FFT) benchmark. The FFT was modeled as a twelve task design and was temporally partitioned into

Table 5.3: **Results of DCT and FFT tested on Wilforce**

Design Name	Device No.	Temporal Partition 1		Temporal Partition 2	
		Tasks	Area(CLBs) (Est., Actual)	Tasks	Area(CLBs) (Est., Actual)
DCT	1	t4_dim1_rows34	(521, 548)	–	–
	2	t1_dim1_rows12	(532, 563)	–	–
	3	t2_dim1_rows34	(536, 563)	–	–
	4	t3_dim1_rows12	(523, 548)	–	–
FFT	1	f2,f3	(285, 264)	g3img	(113, 94)
	2	g3real,g4real	(392, 370)	g4img	(307, 223)
	3	f4,g1real,g1img	(432, 370)	g2img	(285, 228)
	4	f1,g2real	(432, 388)	–	–

two segments. The table lists the latency constraints ($L_{constraint}$) on both temporal partitions and the device area (A_{device}) in the first three columns. For each set of constraints, the models (single-partition and multi-partition) of exploration were performed. The first two sets of constraints represent the experiment for a *XC4013* device with the *minimal* (154,80) and the *maximal* (162,82) latency constraint on each temporal segment. In these cases, both models found a constraint satisfying solution. Nevertheless, it can be seen that the multi-partition exploration finds a better solution (lower design areas).

The next two sets of constraints are with a slightly tighter device area constraint and varying latency constraint on temporal segment TP0. For these cases (and others not shown here), both models of exploration generate identical results. This can be explained as follows. FFT is not as compute-intensive as the DCT example and each FFT task has only two or three possible implementations with little variation in the design area and latency. This provides no leverage to exploration engine to perform a latency-area trade-off on the tasks. Due to the very limited set of available design points, both exploration models converge easily to the best possible solution.

For all the experimental runs, we provided a relaxed memory and interconnection constraint and these were satisfied by all generated solutions. Therefore, the comparative results (solution fitness) directly denotes the ability of the proposed exploration technique to efficiently perform area-latency tradeoff by performing a multi-partition exploration.

5.5.2 Onboard Testing

We modeled the DCT example as a four task design and automatically partitioned and synthesized this for the *Wildforce* board [84], from *Annapolis Micro Systems*. The tasks $t1$ and $t2$ perform the first dimension matrix multiplication of DCT and the tasks $t3$ and $t4$ perform the second dimension. There was no necessity for temporal partitioning since all four tasks fit within the four *XC4013*s on Wildforce. The spatial partitioning and exploration process completed in less than a minute on a 143 Mhz *Sun Sparc-5* processor with 128 MB RAM. Table 5.3 shows results of USM spatial partitioning with exploration.

We automatically partitioned and synthesized the FFT example for the *Wildforce* board [84]. This design was temporally partitioned into two temporal segments with nine tasks in the first segment and three tasks in the second temporal segment. Using the USM partitioning and

exploration environment, both temporal segments were spatially partitioned into the four *XC4013* devices on the board. The spatial partitioning and exploration process completed in 130 seconds on a 143 Mhz *Sun Sparc-5* processor with 128 MB RAM. Table 5.3 shows results of USM spatial partitioning with exploration.

These examples were further synthesized through commercial logic (*Synplicity*) and layout (*Xilinx M1*) synthesis tools to generate the FPGA bitmaps. These design were loaded and successfully executed on the *Wildforce* board. After behavioral modeling, the complete design automation process including simulations and testing using the SPARCS tools was performed within a day, for each example.

5.6 Summary

This chapter describes the tight integration of design space exploration with spatial and temporal partitioning algorithms in the SPARCS system [26]. In particular, this chapter proposes a *spatial partitioning knowledgeable exploration technique* for parallel-process behavioral specifications. The exploration technique has the capability to simultaneously explore the hardware design space of multiple spatial partitions. This enables exploration and spatial partitioning to generate constraint satisfying designs in cases where the traditional exploration model fails. In [69], we introduced the idea of a partitioning-based exploration model for single-process behavioral specifications. In this chapter, we extend the model to handle parallel-process (USM) specifications. Results are presented to demonstrate the effectiveness of the exploration technique and design automation process using SPARCS.

Chapter 6

Light Weight Versions of Existing Synthesis Algorithms

6.1 Introduction

The traditional view of high level or behavioral synthesis (HLS) [124] involves transforming the behavioral specification of a design into a register transfer level (RTL) specification which usually consists of a data path and a controller. The first step in HLS involves extracting an intermediate form from the behavioral specification. The intermediate form is usually a dependency/precedence graph (DG or DFG) representing the behavioral specification. The DFG (data flow graph) is a directed acyclic graph which consists of nodes that represent operations and edges that represent either a control dependency or a data dependency [125]. The RTL design is produced by gradually performing the various synthesis tasks [126] on the DFG.

High level synthesis [124] has gained popularity in the last decade, the key reasons being: (i) Design specifications at higher levels of abstraction are easier to write and allow functionality and design constraints to be clearly stated; (ii) Synthesis algorithms that perform design optimizations have been well established; (iii) HLS allows the designer to explore a large design space in a relatively small amount of time.

Scheduling is an important step in HLS [126, 123, 127]. Scheduling can be described as the process of dividing the DFG into time steps that correspond to clock cycles at the RTL level. Therefore, scheduling directly controls the throughput rate of the RTL design produced. However, for large designs the task of finding optimal schedules is a bottleneck in terms of synthesis time. Therefore, *there exists a tradeoff between the scheduling time and design performance.* A designer would try to exploit this tradeoff using good scheduling algorithms that need to be computationally simple, and at the same time produce high-quality schedules.

Scheduling can be done either under resource constraints (design area and component library) or under time constraints (design speed). A wide variety of algorithms [123, 128, 129, 130] exist in the current literature to perform both kinds of scheduling. In this chapter we are primarily concerned about the Force Directed Scheduling (FDS) algorithm [123] that takes *resource constraints* and tries to optimize the latency (or throughput) of the design. In this section, we will call the resource-constrained FDS algorithm the *Force directed List Scheduling* (FDLS) algorithm. FDLS produces good quality schedules but at the cost of computationally intense force

calculations. Moreover, FDLS would perform poorly without a lookahead into the descendant operation forces, as shown in the results of [131]. Therefore for data flow dominated designs, FDLS is inefficient in terms of scheduling time. *In this chapter, we present a technique to cut-off force calculation of an operation at a certain level of its descendants. This would considerably reduce the scheduling time without degrading the schedule quality.*

In the following section we will present a discussion of related work. In Section 6.3 we will briefly describe the force directed list scheduling algorithm. Section 6.4 presents an example to show the performance of FDLS and an improvement that can be done. Section 6.5 describes the new technique of force calculations based on a concept of stability. Section 6.6 describes how the stability concept can be extended to any DFG, using stability conditions. It also presents the FDLS algorithm that is coupled with a stability condition. Finally, in Section 6.7, we present detailed results for a suite of high level synthesis benchmarks. The results show a clear improvement in the performance of FDLS when coupled with the stability condition.

6.2 Related Work

In the past there have been improvements suggested to the Force-Directed Scheduling (FDS) algorithm. Here we will compare our improvement with some of these. We assume that the reader is familiar with Paulin and Knight's FDS algorithm [123].

Verhaegh et al. in [132] proposed an improvement to the time-constrained FDS algorithm (that minimizes resources within a given schedule length), using a gradual time frame reduction technique. Instead of directly scheduling an operation to a time step within its time frame (as suggested by Paulin and Knight in [123]), the time frame for the chosen operation is reduced by one time step. Our implementation of the resource-constrained FDS algorithm follows a very similar approach. At each time step, if an operation is not chosen for scheduling then its time frame is reduced by one time step. The only difference is that we use a more rigorous force function. The authors in [132] also suggest a way of computing the force of an operation using a requirement distribution function similar to the one suggested by Paulin and Knight. Their force of scheduling an operation at a time step is computed by summing the changes in the probabilities of *all* the operations. Operations whose time frames have not been affected would contribute a zero force. In our implementation of the resource-constrained FDS algorithm we only accumulate the forces of descendants of the operation, since only their time frames would be affected.

Verhaegh et al. in [133] propose an incremental way of computing forces, to reduce the complexity of the time-constrained FDS algorithm. Initially, the set of operations whose time frames have changed is determined. Then depending on whether an operation belongs to this set, they either compute the force by summing only the changes in other forces or recompute the entire force. As mentioned earlier, our implementation of the resource-constrained FDS algorithm computes forces only for those operations whose time frames have changed, but we have not done incremental computation.

We observe that none of the improvements mentioned earlier suggest the possibility that the forces of certain descendant operations need not be summed up even though their time frames might have changed. The technique suggested in this chapter dynamically determines the number of descendant operations whose forces have to be summed to get the force of an operation. The improvement suggested in this chapter is independent of those suggested in the past and can be used in conjunction with them for further improvement. Since our implementation of the

```
Force_Directed_List_Scheduling(DFG, R_set)
Begin
   T_max ← Critical Path Length in the DFG
   T_step ← 1
   while (T_step ≤ T_max)  ▷ Each iteration corresponds to a T_step
      Evaluate Time Frames
      L_ready ← { All operations whose time frames intersect with T_step}
      while (R_set not sufficient)  ▷ Need to defer an operation
         if (all operations in L_ready are on critical path) then
            T_max ← T_max + 1
            Evaluate Time Frames
         end if
         Compute_Deferral_Forces()
         Op ← Operation in L_ready with the least force
         L_ready ← L_ready − {Op}  ▷ Defer the operation
      end while
      for each (operation Op ∈ L_ready)
         Schedule Op at T_step
      end for
      T_step ← T_step + 1
   end while
End
```

Figure 6.1: **Force Directed List Scheduling Algorithm**

improved resource-constrained FDS algorithm already incorporates some of the improvements suggested in the past, the speed-up results shown in this chapter justify the fact that the improvement achieved by using our technique is independent of others.

6.3 Force Directed List Scheduling

The FDLS (resource-constrained FDS) algorithm proposed by Paulin and Knight [123] is a very popular technique of scheduling, widely used in many synthesis tools that exist in the current literature. The FDLS algorithm shown in Figure 6.1, is based on the well known *list scheduling algorithm* [134]. Operations are sorted in a topological order based on the control and data dependencies extracted from the DFG. At each time step ($\mathcal{T}_{step}$) a list of operations that are ready to be scheduled, called the *ready list* ($\mathcal{L}_{ready}$) is formed. As long as the resource set ($\mathcal{R}_{set}$) is not sufficient to schedule the operations in the ready list the inner-while loop (See Figure 6.1) keeps deferring an operation in each iteration. In order to select an operation to defer, a deferral force is calculated for each of the ready list operations and the least force operation is picked. When the resources are sufficient the remaining operations in the ready list are scheduled in the current time step.

There are two main tasks in each iteration (time step) of FDLS, namely, the evaluation of the time frames and computation of the deferral forces for each operation in the ready list. The time frame of an operation is specified by its As Soon As Possible (ASAP) and As Late As Possible (ALAP) time steps. The evaluation of a time frame would simply involve updating of these two values. On the other hand, a deferral force calculation involves updating of the distribution graph [123] of the operation and computation of the deferral force of the operation. The force of

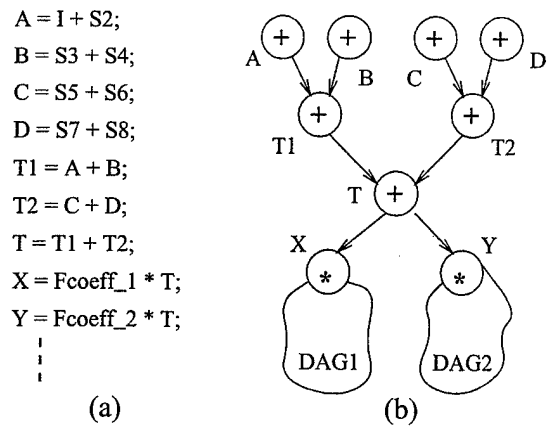


Figure 6.2: **(a) Filter Behavioral Specification (b) The DFG**

deferring an operation to a time step (i), as suggested by Paulin and Knight [123] is [1]:

$$\text{Force(i)} = \text{Self-Force(i)} + \text{Successor\_Forces}$$

The self-force of an operation determines the average competition offered by those operations that appear in its time frame and compete for its resources. The successor_forces are computed by summing up the self-forces of all the descendants whose time frames would be modified due to the deferral of the parent operation. Note that, a force calculation is a more computationally intensive task than the evaluation of a time frame. More importantly, note that the deferral forces are computed for all the operations in the ready list, in each iteration of the inner-while loop (Figure 6.1). The time frames of the operations are not necessarily evaluated in each iteration. In our experience, we have observed for many design examples that *the most computational intensive task of the FDLS algorithm is the calculation of successor forces.*

6.4 Motivation through an Example

In this section, we present an example to show that FDLS is not efficient in its successor force calculations. When resources are insufficient, FDLS computes forces in order to select an operation for deferral. As described earlier, when computing the force of an operation, FDLS also computes the self-forces of all the descendants of that operation. However, it may not always be necessary to compute the forces of *all* the descendants, in order to make an effective deferral decision.

Consider the portion of a behavioral specification for a filter example shown in Figure 6.2.a and the corresponding DFG in Figure 6.2.b. The previous state variables (s2 through s8) and the filter input (I) are summed up and used in the computation of the next state values and the filter output, using the filter coefficients (Fcoeff_1 and Fcoeff_2). This computation is done in the two DAG fragments (DAG1 and DAG2), as shown in Figure 6.2.b. The operations (nodes) in the DFG are marked by the corresponding variable names in the behavioral specification. Consider a resource set that has only one adder. At the first time step, the ready list has the four addition operations A, B, C and D. In order to pick an operation for deferral, FDLS computes the forces of all these operations and then defers the least force operation.

[1] See [123] for a detailed explanation of this function. We assume that the reader is familiar with [123]

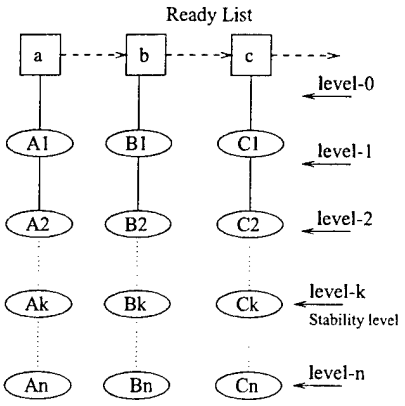


Figure 6.3: **Topological ordering of ready operations and their descendants**

When computing the successor force of operations A, B, C and D, FDLS goes all the way down till the end of DAG1 and DAG2, and computes the self forces of all their descendants. But an important point to note here is that the operation T is a *common descendant* of the four addition operations in the ready list. This implies that all descendants of A, B, C and D that are also descendants of T, would equally contribute to the force of the operations A, B, C and D. For example, the self-force of the descendants X and Y would be added to the force of all the four addition operations since X and Y are also descendants of T. Therefore addition of the self-forces of X and Y and any other descendants of T, would not change the selection of one of the four additions for deferral. In other words, we can state that if there exists a *common descendant* for the ready list operations and all other descendants are either predecessors or successors of the common descendant (i.e, this is a *bottle neck* descendant), it is enough if we compute self-force of those descendants that are predecessors of the common descendant, including the common descendant. In this example, while computing successor forces of A, B, C and D, it is enough if the self-forces of the operations T1, T2 and T are computed, since T1 and T2 are the only descendants that are predecessors of the *closest common descendant* T. This observation can be suitably extended to the case of a set of common descendants.

6.5 The Stability Concept

In the previous section we presented an example to show that successor force computation can be limited to a subset of descendants of the ready list operations. However, the concept cannot be easily extended to all DFGs, since there may not be a common descendant for all the operations in the ready list. In such a case, we would have to find common descendants for mutually exclusive partitions of the ready list. Even then, we can only state that it is not necessary to compute successor forces beyond the closest common descendant for operations in that partition of the ready list. However, the actual deferral operation could belong to any partition. Therefore, successor force computation, beyond such common descendants may still affect the deferral decision. The problem is therefore not solved.

We can however look at the problem from a different point of view. Consider a ready list shown in Figure 6.3. The operations and their descendants are topologically sorted into levels. The list of ready operations (a, b, c ..) are considered to be in level-0. Their immediate successors are considered to be in level-1. For example, A1 is the set of immediate successors of the operation

'a', B1 has the immediate successors of the operation 'b' and so on. Similarly, A2 is the set of all immediate successors of each of the operations in the successor set A1. Therefore A1 is defined as the parent set of A2. We recursively define *an ith level successor set of an operation in the ready list to be the set of immediate successors of all the operations in the parent set at the (i-1)th level.* Also for any ith level successor set (Ai) we define a corresponding *parent operation* (a) in the ready list. If the ith level successor set of any operation in the ready list is empty, then successor sets at all levels greater than i are also empty.

The computation of successor forces for the operations in the ready list can now be done level by level. At each level-i starting from the first, the self-forces of all the operations in the ith level successor set is added to the corresponding parent operation in the ready list. Let n be the farthest level of successors for any operation in the ready list. FDLS would compute successor forces at each level, until level-n and pick the least force operation for deferral. If we sort the ready list operations in the order of their force values, it is very well possible that there exists a level-k (k less than n) after which this order *stabilizes*, i.e, does not change. We call this level-k as the *stability level.*

For example, if there exists a common descendant for the ready list operations then the stability level is level-k where k is the length of the path from any ready operation to that common descendant. Due to the common descendant, after this level-k the addition of successor forces would equally contribute to the forces of all the operations in the ready list, thereby preserving the order of the operations. There could be several other reasons why this stability level can exist, a few of them are: (i) Successor forces after a certain level k might become zero if the successors after level-k have similar distribution graphs; (ii) After level-k, the successors may contribute the same force to each of the ready operations. This could be true if successors at level-k have similar time frames and equal competition; (iii) When there is no single common descendant for all operations in the ready list, but only for mutually exclusive partitions of the ready list, the stability level could still exist. This would be true if the addition of successor forces does not change the ordering of operations between the partitions of the ready list.

The problem at our hand is to find this stability level. If we can dynamically detect the stability level then we can stop computation of successor forces at that level and pick the least force operation for deferral. *Since the relative ranking (with respect to forces) of the operations at the stability level is the same as what it would be after all the n levels of successors force calculations, the operation selected for deferral would also be the same. Therefore, we can be assured that the schedule quality would be preserved.*

6.6 FDLS that uses a Stability Condition

In order to dynamically (at run time) detect the stability level in each iteration of FDLS, we have to look at the forces of the ready list operations after each level of successor force computation. We now define a *stability condition* (SC) as a condition which when true indicates that stability level has been reached. The stability condition will be used in the FDLS to dynamically detect the stability level and stop successor force computation. Table 6.1 presents three simple stability conditions that have their own advantages and disadvantages.

In Table 6.1, the stability condition SC-1 checks if the force value of any one of the ready list operations has changed between two consecutive levels of successor forces computation. The intuition is that, if none of the force values of the ready list operations have changed, then the

Table 6.1: **Some simple stability conditions**

S.No	Stability Condition	Sorting Required
SC-1	Force of any operation changed.	No
SC-2	For each Op_type, least force Operation Set is preserved.	Yes
SC-3	Top N forces of each Op_type remain unchanged.	Yes

order of operations is preserved, identifying stability. SC-2 works based on a different approach to stability. At each level, there is a set of least force operations, since more than one operation can have the least force value. SC-2 detects stability if this least force operation set remains the same overs two levels of successor force computation. This intuition is that, if the set of least force operations remain the same then it is highly possible that one of them is the actual operation that would have been selected for deferral, after all levels successor force computation. SC-3 on the other hand looks at the other extreme. It checks if the forces of the top N operations remain fixed, i.e, force values did not change. Here N is the number of resource of this operation type available. The ready list is assumed to be organized as a collection of operation types and operations belonging to each type. Intuitively, if the top N forces remain fixed, then there is a high probability that one of these operations is chosen for scheduling in this time step. Note that SC-2 and SC-3 check individually for each operation type in the ready list and require sorting after each level of successor force computation. SC-1 is too strong a stability condition and therefore might miss some cases of actual stability. On the other hand SC-2 and SC-3 are relatively more tolerant in stability detection. Each of these stability conditions is only a heuristic and therefore may fail to detect stability in some cases. But it is important to see how often such cases appear in real life design examples, which can only be seen by experimentation.

We have constructed a stability condition that takes the best aspects of the three conditions presented above. Figure 6.4 shows the procedure for deferral force computation in FDLS with our stability condition plugged in. The figure also shows the computation of successor forces level by level, starting from the first. The initial for-loop computes forces of ready operation at level-0. Each iteration of the second for-loop corresponds to a successor level. The body of second for loop has two parts: (i) the computation of successor forces at the current level and (ii) the stability condition, as shown by the commented lines in Figure 6.4.

Our stability criterion checks for two conditions: (1) Order of all the operations (relative ranking with respect to forces) is preserved, and (2) At least one force value does not change for each operation type in the ready list. The first condition ensures that the same order of operations is preserved over two consecutive successor levels but is too weak by itself. For example, if the forces of all operations were to change by a constant value, the order is preserved. But this might not be point of the stabilization of the order because all the force values have just changed and there is a probability that some of them might change in the next level, leading to instability. However, in conjunction with the second condition it would not detect stability since none of the force values remained the same. Therefore, adding the second condition makes it a bit more tolerant. Also we have added one more condition to make it efficient in sorting. Sorting is done only when any of the forces have changed their values. Finally, we have the Req_Csl (Required number of Consecutive Stability Levels) criterion that adds a user specified level of tolerance to the

```
Compute_Deferral_Forces()
Begin
   for each (operation Op ∈ L_ready)
      Op.force ← Self_force of Op
   end for
   Sort operations based on their forces
   Req_Csl ← Required number of consequetive stability levels
   Nsl ← 0   ▷ Number of stabilized levels
   for (succ_level ← 1 to Critical_Path_Length) do
      ▷ COMPUTATION OF SUCCESSOR FORCES
      for each (operation Op ∈ L_ready)
         Determine Successor_Set at 'succ_level'
         succ_force ← Sum of Self forces of all Successor_Set operations
         Op.force ← Op.force + succ_force
      end for
      ▷ STABILITY CONDITION
      Order_Changed ← False
      if (any force value changed) then
         Sort operations based on their forces
         if (any change occurred in the order) then
            Order_Changed ← True
         end if
      end if
      if (Order_Changed) then
         Nsl ← 0   ▷ Reset. Instability before Req_Csl reached
      else
         if (for each op_type, at least one force did not change) then
            Nsl ← Nsl + 1   ▷ One more level of stability reached
         else
            Nsl ← 0   ▷ Reset. Instability before Req_Csl reached
         end if
      end if
      if (Nsl = Req_Csl) then
         break   ▷ STOP. Desired level of stability reached
      end if
   end for
End
```

Figure 6.4: **Dynamic Successor Force Computation in FDLS**

Table 6.2: **Information on the synthesis benchmarks used**

Design Example	Operation information from DFG	Total Operations	Design Space	Critical Path Len.
1. EWF	$\{26 : [+],\ 8 : [*]\}$	34	36	15
2. LSS	$\{12 : [+],\ 16 : [\div],\ 16 : [*]\}$	44	3072	4
3. DCT4x4	$\{96 : [+],\ 128 : [*],\ 16 : [\div]\}$	240	49152	8
4. TN-1	$\{416 : [+],\ 512 : [*],\ 96 : [\div]\}$	1024	$3.6 * 10^6$	14
5. TN-2	$\{640 : [+],\ 768 : [*],\ 128 : [\div]\}$	1536	$2.9 * 10^7$	9
6. DCT8x8	$\{896 : [+],\ 1024 : [*],\ 64 : [\div]\}$	1984	$1.4 * 10^7$	10

condition. The condition can be tuned to check for stability over 'Req_Csl' number of consecutive successor levels. We have compared the performance of FDLS with and without this stability condition. Results presented in the following section show considerable improvement in execution time for the same schedule quality, for different synthesis benchmarks.

6.7 Results

In this section we will compare the performance of the two versions of FDLS: with and without the stability condition. For obtaining the results, we have considered a suite of high level synthesis benchmarks, that is shown in Table 6.2. We use a library of parameterized RTL components using which the scheduler generates resource sets for a given design. For each example in the table, we have shown the operation information, the total number of operations, the number of generated resource sets (design space) and the critical path length in the DFG.

The design examples shown in Table 6.2 are listed in the order of increasing sizes (number of operations in the DFG). Our first example, the smallest in the suite, is an Elliptic Wave Filter (EWF) taken from Kung, Whitehouse and Kailath's book on signal processing [136]. It contains 34 operations that are subjected to over fifty precedence constraints. This was also chosen as a benchmark for the 1988 High-level synthesis workshop. Our largest design example (DCT8x8), computes the Discrete Cosine transform (DCT) [135] of an 8x8 matrix. DCT8x8 has a total of 1984 operations in the DFG, out of which 896 are additions, 1024 are multiplications and 64 are divisions. It has a design space consisting of more than ten million resource sets. The second example, is a Linear System Solver (LSS) [137], which is a popular method of solving a linear system of equations using matrix inversion. This LSS example computes the solution to a system of four equations with four variables each. The third example (DCT4x4), is a smaller version of DCT that computes the transform of a 4x4 matrix. The Threshold Network (TN) [138] is widely known as the perceptron network in neural systems. Each node in the network takes a variable number of inputs and generates the average of their weighted sum as the output. We have considered two versions of the threshold network (TN-1 and TN-2), each having nodes that take either four or eight inputs, but the number of nodes and their connectivity are different.

Table 6.3 compares the execution times of the two FDLS versions: the original FDLS and our Dynamic-FDLS (D-FDLS) that dynamically cuts off successor force computation using the stability condition presented in Section 6.6. We have measured the execution times using the commercial Quantify tool. The results have been taken on a Sparc-20 machine with a 256 Megabyte memory and a clock speed of 75 MHz. Note that, for the smaller examples the

Table 6.3: **Execution times for FDLS and Dynamic FDLS**

Design Example	Resource Set Information	Execution Time			Schedule Length	
		FDLS	D-FDLS	Reduction(%)	FDLS	D-FDLS
1.EWF	{1 : [+], 1 : [−]}	9.43ms	8.23ms	12.7	21	21
	{1 : [+], 2 : [−]}	9.42ms	8.17ms	13.2	21	21
	{1 : [+], 3 : [−]}	6.85ms	5.50ms	19.6	21	21
	{2 : [+], 1 : [−]}	4.72ms	4.15ms	11.9	16	16
	{2 : [+], 2 : [−]}	4.67ms	4.12ms	11.7	16	16
	{3 : [+], 2 : [−]}	2.96ms	2.96ms	0	15	15
2.LSS	{3 : [+], 1 : [÷], 4 : [*]}	33.03ms	25.04ms	24.2	19	19
	{3 : [+], 2 : [÷], 4 : [*]}	17.65ms	13.50ms	23.5	13	13
	{6 : [+], 2 : [÷], 8 : [*]}	17.65ms	13.46ms	23.7	11	11
	{3 : [+], 4 : [÷], 4 : [*]}	10.11ms	7.82ms	22.7	10	10
	{6 : [+], 4 : [÷], 8 : [*]}	9.54ms	7.32ms	23.4	8	8
	{6 : [+], 8 : [÷], 8 : [*]}	4.54ms	3.55ms	22.0	6	6
	{12 : [+], 8 : [÷], 16 : [*]}	4.47ms	3.47ms	22.5	5	5
	{12 : [+], 16 : [÷], 16 : [*]}	1.63ms	1.63ms	0	4	4
3.DCT4x4	{3 : [+], 4 : [*], 16 : [÷]}	2489.29ms	869.96ms	65.05	39	39
	{6 : [+], 8 : [*], 16 : [÷]}	1041.93ms	368.46ms	64.64	23	23
	{12 : [+], 16 : [*], 16 : [÷]}	396.8ms	125.47ms	68.38	15	15
	{18 : [+], 24 : [*], 16 : [÷]}	233.05ms	63.69ms	72.71	12	12
	{24 : [+], 32 : [*], 16 : [÷]}	158.92ms	45.11ms	71.61	10	10
	{36 : [+], 48 : [*], 16 : [÷]}	114.62ms	36.9ms	67.81	10	10
	{48 : [+], 64 : [*], 16 : [÷]}	24.65ms	24.65ms	0	8	8
4.TN-1	{7 : [+], 8 : [*], 2 : [÷]}	338.63s	167s	50.68	91	91
	{14 : [+], 16 : [*], 4 : [÷]}	129.43s	57.89s	55.27	47	47
	{28 : [+], 32 : [*], 8 : [÷]}	35.13s	3.11s	91.1	25	25
	{56 : [+], 64 : [*], 16 : [÷]}	16.19s	0.55s	96.6	17	17
	{112 : [+], 128 : [*], 32 : [÷]}	1.21s	0.28s	76.53	15	15
	{224 : [+], 256 : [*], 64 : [÷]}	0.4s	0.4s	0	14	14
5.TN-2	{7 : [+], 8 : [*], 2 : [÷]}	310.63s	199s	35.94	107	107
	{14 : [+], 16 : [*], 4 : [÷]}	141.2s	88.07s	37.63	55	55
	{28 : [+], 32 : [*], 8 : [÷]}	58.19s	30.91s	46.88	29	29
	{56 : [+], 64 : [*], 16 : [÷]}	11.68s	2.98s	74.49	16	16
	{112 : [+], 128 : [*], 32 : [÷]}	9.63s	2.33s	75.8	12	12
	{224 : [+], 256 : [*], 64 : [÷]}	1s	0.67s	33	10	10
	{448 : [+], 512 : [*], 128 : [÷]}	0.91s	0.91s	0	9	9
6.DCT8x8	{7 : [+], 8 : [*], 64 : [÷]}	1335.27s	288.33s	78.22	140	140
	{14 : [+], 16 : [*], 64 : [÷]}	621.62s	135.38s	78.41	76	76
	{21 : [+], 24 : [*], 64 : [÷]}	389.65s	88.50s	77.29	50	50
	{28 : [+], 32 : [*], 64 : [÷]}	270.84s	60.36s	77.71	44	44
	{42 : [+], 48 : [*], 64 : [÷]}	233.37s	49.19s	78.92	31	31
	{56 : [+], 64 : [*], 64 : [÷]}	110.24s	24.67s	77.62	28	28
	{84 : [+], 96 : [*], 64 : [÷]}	65.61s	15.92s	75.73	19	19
	{112 : [+], 128 : [*], 64 : [÷]}	43.97s	8.14s	81.48	16	16
	{384 : [+], 128 : [*], 64 : [÷]}	44.57s	8.77s	80.32	13	13
	{224 : [+], 256 : [*], 64 : [÷]}	19.81s	2.55s	87.13	12	12
	{448 : [+], 512 : [*], 64 : [÷]}	1.43s	1.43s	0	10	10

Table 6.4: **Total Execution Time saved during Design Space Exploration**

Design Example	Av. Exec. Time	Av. % Reduction	Av. Time Saved	Total Res. Sets	Total Time Saved
1. EWF	6.2ms	13.8	0.855ms	36	30.78ms
2. LSS	17.33ms	23.1	4.003ms	3072	12.23s
3. DCT4x4	1301.95ms	68.3	890.01ms	49152	12hrs 9min
4. TN-1	169.92s	74	125.74s	$3.6 * 10^6$	14 years
5. TN-2	155.82	50.6	78.84s	$2.9 * 10^7$	72 years
6. DCT8x8	668.35s	79.3	530s	$1.4 * 10^7$	235 Years

execution times are in the order of milliseconds and for the larger examples, they are in the order of seconds. We have chosen resource sets at evenly spaced design points in the design space of each example. The resource sets for each of these examples are listed in the order of increasing areas, starting from the smallest area resource set. For each entry we show the quantity and function of each resource used. Note that for all the examples the schedule length produced by FDLS with the stability condition is the same the one produced by the original FDLS. The stability condition proposed is Section 6.6 has not failed in any of these cases. For all the examples, we had set the number of consecutive levels of stability (Req_Csl) as 1. Note that if there is no stability point, the stability condition would let FDLS compute successor forces till the last successor level thereby producing the same schedule. For the last four examples, FDLS with the stability condition was able reduce at least 50% (on an average) of the execution time. This implies two points: (i) There does exist a stability level for a considerable number of time steps and for all possible resource sets; (ii) Most of the execution time in FDLS is spent in the computation of successor forces. For the EWF and the LSS examples which are comparatively smaller, FDLS with the stability condition still was able to reduce on an average, 13% (for EWF) and 23% (for LSS) of the execution time. Note that for the largest possible resource set of each of these examples, the execution times are identical since forces are not computed at any time step; as the resource sets are sufficiently large, no deferrals are required.

Table 6.4 shows the average execution time, average percentage reduction in execution time and the average time saved for any module set. Since evenly spaced design points were chosen (most parallel, most serial and in between), we can say that the average value is reasonably accurate. If a synthesis tool were to completely explore the entire design space of these examples over all possible resource sets (shown in the fifth column of the table) then the amount of time saved in generating all the schedules is given in the last column. Even for the TN-1 example which is much smaller than DCT8x8, the amount of scheduling time saved is in the order of years. Although, high level synthesis tools do not explore design space exhaustively, this clearly shows that the stability condition coupled with FDLS would be highly useful in better space exploration.

It is possible that in some cases our algorithm cannot find a stability level for any deferral decision either because our stability criterion could not discover stability or because there exists no stability level due to the structure of the DFG. In such cases our algorithm incurs a overhead in execution time due to the need for re-sorting the ready list when operator forces change. This overhead is negligible due to efficient implementation of the re-sorting operation.

6.8 Conclusion

The widely used Force Directed List Scheduling spends a major portion of its execution time in the calculation of successor forces. In this chapter, we have presented a concept of stability by which successor force computation in FDLS can be dynamically cut-off thereby leading to a reduction in execution time. The stability level is identified with the help of a stability condition. In general, it is very difficult to come up with a single stability condition that would work for all kinds of DFGs. In fact, a version of FDLS could also use multiple stability conditions for different time steps of the schedule. We have shown three different stability conditions and formed one which takes the best aspects of these, and plugged it into FDLS. Results presented in this chapter show that this stability condition results in considerable reduction in the execution time of FDLS for the same schedule quality, for a suite of high level synthesis benchmarks.

Chapter 7

RC and FPGA FloorPlanning

7.1 Introduction

Placement and floorplanning are extensively studied topics. However, the importance of placement and floorplanning cannot ever be ignored due to changing design complexities and requirements. Currently, commercially available devices can map up to one million gate equivalent designs[29] (and some of the newly announced products like Altera's APEX series will map over two million gate equivalent designs[28]). Such complex design densities also demand tools that can efficiently and quickly make use of available gates.

Improvements in CAD tools for FPGAs have not kept pace with hardware improvements. The available tools typically require from minutes to hours to map[1] designs (or circuits) with just a few thousand gates, and as design sizes increase the execution time will increase. One way to address the problem of long mapping times is create designs that use premapped macros[2] to create larger designs (macro based circuits). Then, floorplan and route these macro based circuits. This approach combines the technology mapping and physical placement steps of the circuit mapping process. In general, floorplanning is an NP-hard problem [73]. For FPGAs, it is more difficult due to fixed logic resources. Additionally, the netlist is not always complete. In certain instances (for example high level synthesis) the circuit netlist may not contain any interconnect information. In this case, the floorplanning problem reduces to the two-dimensional packing problem.

To address the problem of mapping large designs to large FPGA circuits, we have taken a macro based approach [16, 17]. When a complete netlist (set of macros and interconnects) is available, we floorplan interconnected macro based circuits. For the case where we have no interconnect information or there are problems making the circuit fit the available area, we perform two-dimensional packing on the set of macros. At the lowest level, a macro is composed of one or more interconnected and relatively placed logic blocks. In this dissertation, we present a method (based on clustering and tabu search (TS) optimization) to quickly floorplan interconnected, macro based circuits (circuits composed of interconnected macros) while attempting to minimize throughput delay and meet area and routability constraints. For the case when interconnect information is unavailable or there are problems making the circuit fit the given area, we present a method (based on grouping soft and hard macros) to quickly floorplan the set of macros making

[1] typical mapping steps include design entry, technology mapping, placement, and routing

[2] macros are predefined circuit components like adders, shifters, decoders, multipliers, signal processors, CPUs, etc.

up the circuit.

The basic flow of our method is summarized as follows. We start with a set of macros (M) interconnected by a set of signals (S). Each macro is composed of a set of interconnected relatively placed logic blocks. If $\mid S \mid > 0$, we then group (cluster) macros together to form clusters. Each cluster in the set of clusters (B) is smaller in area than some predefined limit[3]. We then use TS optimization to perform two-dimensional placement on the set of clusters B. Then, for each cluster that is composed of more than one macro, we perform intracluster placement[4]. Finally, for any macro whose shape was changed during the intracluster placement process, we perform intramacro placement[5].

In the event $\mid S \mid = 0$ or the set M will not fit using the process described above, we separate the set M into two non-overlapping sets of soft and hard macros. We then perform two-dimensional packing on the set of hard macros. If the hard macros fit the given area, we go back and floorplan the soft macros and change their shape as necessary.

In this chapter we present our floorplanning methodology. In section 7.2 we formally describe the floorplanning problem and lay the foundation for the solution. In section 7.3 we describe our tabu search (TS) based floorplanner and methodology. In section 7.4 we explain our test methodology, and in section 7.5 we analyze the data. Finally in section 7.6 we provide conclusions for our methodology.

7.2 Floorplanning Problem

Given a set of macros $M = \{m_1, m_2, ..., m_n\}$ and a set of signals $S = \{s_1, s_2, ..., s_q\}$, we associate with each macro $m_i \in M$, a size a_i (number of logic blocks in m_i); a width w_i (maximum width of m_i in number of logic blocks); a height h_i (maximum height of m_i in number of logic blocks); a flexibility f_i (0 for hard/fixed macros or 1 for soft/flexible macros); and a set of interconnecting signals S_{m_i} ($S_{m_i} \subseteq S$). For hard macros (macros with fixed size, shape, and internal placement), w_i and h_i are both fixed and $f_i = 0$. For soft macros (macros with fixed size and variable shape), w_i and h_i are considered flexible (both w_i and h_i can take on a range of values typically between 1 and a_i) and $f_i = 1$. Additionally, for the case when $\mid S \mid > 0$, with each signal $s_i \in S$ we associate a set of macros M_{s_i} where $M_{s_i} = \{m_j \mid s_i \in S_{m_j}\}$. M_{s_i} is said to be a *signal net*. We can divide M into two distinct sets, MS and MH (subset of soft macros and subset of hard macros), where $M = \{\ MH \cup MS \mid MH \cap MS = \phi,\ f_i = 0\ \forall\ m_i \in MH$, and $f_i = 1\ \forall\ m_i \in MS\ \}$. We are also given a target set $L = \{l_1, l_2, ..., l_p\}$ of locations where $\mid L \mid \geq \sum_{i=1}^{\mid M \mid} a_i$. For the case of mapping $m_i \in M$ to a regular two-dimensional array, each $l_j \in L$ is represented by a unique (x_j, y_j) location[6] on the surface of the two-dimensional array where x_j and y_j are integers. Additionally, we define the two-dimensional array L by the width of physical logic block locations, W_L, and the height of physical logic block locations, H_L. Figure 7.1 shows the 16 element set L for an example 4×4 two-dimensional array ($W_L = 4$ and $H_L = 4$). When $\mid S \mid > 0$, the floorplanning problem becomes how to assign each soft macro $m_i \in MS$ a shape and each macro $m_j \in M = MH \cup MS$ a unique location in L such that an objective function is

[3]predefined limit implies the total area of each cluster (sum of the areas of the macros within the cluster) is less than some maximum

[4]intracluster placement is the task of assigning the macros that make up the cluster a physical location and reshaping any macro whose shape must be altered to meet area constraints

[5]intramacro placement is the process of relatively placing the logic blocks that make up a macro component

[6]for our application, the location represents a physical logic block location on the FPGA

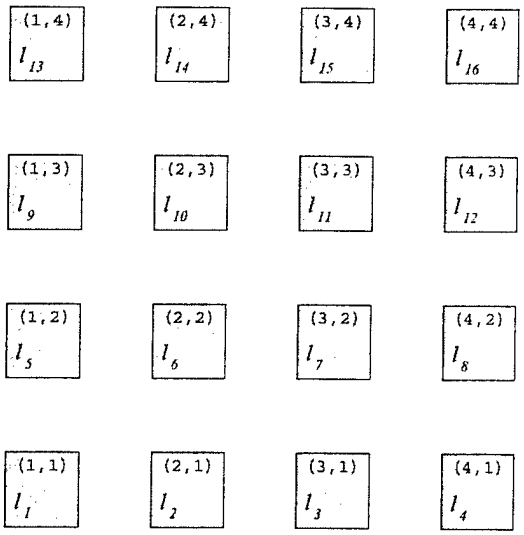


Figure 7.1: **Example two-dimensional array $L = \{l_1, l_2, ..., l_{16}\}$ of physical logic block locations ($W_L = 4$ and $H_L = 4$). One logic block can be assigned to each physical location $l_i \in L$**

optimized. (Here, uniqueness implies no macro overlaps.) In this case, our goal is to optimize the floorplanned circuit's performance while meeting area and routing constraints. When $\mid S \mid = 0$, the floorplanning problem becomes how to assign each soft macro $m_i \in MS$ a shape and each macro $m_j \in M = MH \cup MS$ a unique location in L such that area constraints are satisfied.

7.3 Solution

In this section, we give an overview of our method, and in following subsections we describe each step in detail. First, some preliminary definitions are required. As stated earlier, a macro is a set of one or more interconnected and relatively placed logic blocks. We are given a set M of macros in our circuit or design netlist. When necessary, we group macros in M together to form clusters. Therefore, we define a cluster as a set of one or more macros, and $B = \{b_1, b_2, ..., b_p\}$ as the set of all clusters. (For initialization, there is a one to one mapping of the elements of the set M to the elements of the set B, and therefore, initially $\mid B \mid = \mid M \mid$.) As stated earlier, we are floorplanning the set of macros M on the two-dimensional array L of physical logic block locations. Once macros are grouped to form clusters, our approach is to perform two-dimensional placement of clusters on L. To perform this placement, we divide our target two-dimensional array L into a two-dimensional array of buckets where each bucket (of physical logic block locations) has the same size and shape. (We define the bucket size by a width of W_B logic blocks and a height of H_B logic blocks.) We define the set of buckets as the set $\{l'_1, l'_2, ..., l'_m\} = L'$, where the number of buckets m equals $\mid L' \mid$. (The two-dimensional array L' is defined by a width of $W_{L'}$ buckets and a height of $H_{L'}$ buckets.) Then, instead of performing two-dimensional placement of clusters directly on L, we perform two-dimensional placement of clusters on the smaller set L'. Figure 7.2 shows the example L divided into four equally sized buckets of physical logic block locations where each bucket is 2 logic blocks $\times$ 2 logic blocks.

Figure 7.3 shows a flow chart of our floorplanning methodology. In Figure 7.3, we read in the sets M, S, and L. If $\mid S \mid = 0$, we proceed to the two-dimensional packing stage. If $\mid S \mid > 0$, we

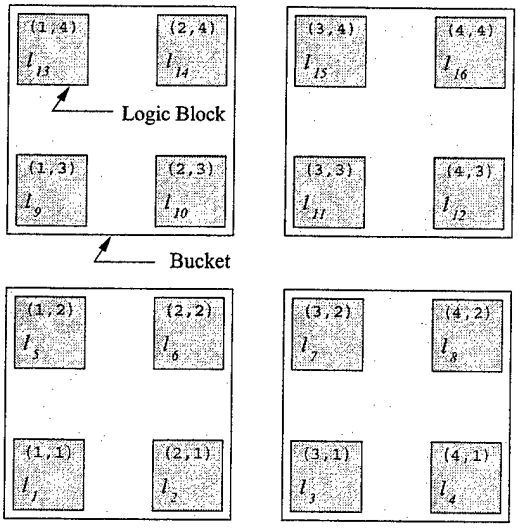


Figure 7.2: **Example L divided into a set L' of 4 buckets. The dimensions of L' are $W_{L'}$ = 2 buckets and $H_{L'}$ = 2 buckets. The dimensions of the example bucket are W_B = 2 logic blocks and H_B = 2 logic blocks**

initialize the set of clusters B. Initially, each element of B contains one element from M, so there is a one to one mapping of the elements of M to the elements of B. After initialization of B, we initialize the bucket width, W_B, and bucket height, H_B, using the procedure Create_Buckets(M). Details for Create_Buckets(M) are found in subsection 7.3.1. After bucket size initialization, we create the set of buckets, L', as outlined in subsection 7.3.2. Next, we check the fit of B on L'. (It should be noted that we create and maintain the bucket width W_B and bucket height H_B so any single macro in M will fit in any bucket in L' [7]. This allows us to skip the clustering step if $\mid B \mid$ is less than or equal to $\mid L' \mid$. This usually occurs when low device utilization is sufficient and allows for very fast floorplanning.) If there is a fit, we proceed to the placement phase; if there is not a fit we proceed to the clustering phase.

In Figure 7.3, if $\mid B \mid$ is not less than or equal to $\mid L' \mid$, we proceed to the clustering phase. To ensure fit, our methodology requires $\mid B \mid$ is less than or equal to $\mid L' \mid$, and therefore, the goal of the clustering phase is to group smaller macros together thereby reducing $\mid B \mid$ until $\mid B \mid$ is less than or equal to $\mid L' \mid$. Additionally, it is required that each cluster $b_i \in B$ has size less than or equal to the bucket size. This ensures each cluster will fit in any bucket. The details of clustering are found in subsection 7.3.3. After clustering if $\mid B \mid$ is less than or equal to $\mid L' \mid$ then we proceed to the placement phase else we iteratively increase the bucket size (as described in subsection 7.3.4) and continue clustering until $\mid B \mid$ is less than or equal to $\mid L' \mid$ or the bucket size exceeds the dimensions of L.

In Figure 7.3, if $\mid B \mid$ is less than or equal to $\mid L' \mid$, we proceed to the placement phase, otherwise, we proceed to the two-dimensional packing phase. In the placement phase we use TS based placement to assign each $b_i \in B$ to a bucket (see subsection 7.3.5). Then, in intracluster placement, we assign each macro within each cluster a physical location and shape (see subsection 7.3.6). Finally, in intramacro placement, we place the logic blocks within any soft macro whose shape has been altered during intracluster placement (see subsection 7.3.7). After this phase, every logic block making up the circuit or design netlist will have a physical location on the

[7] The initial bucket size is based on the dimensions of the largest elements of M.

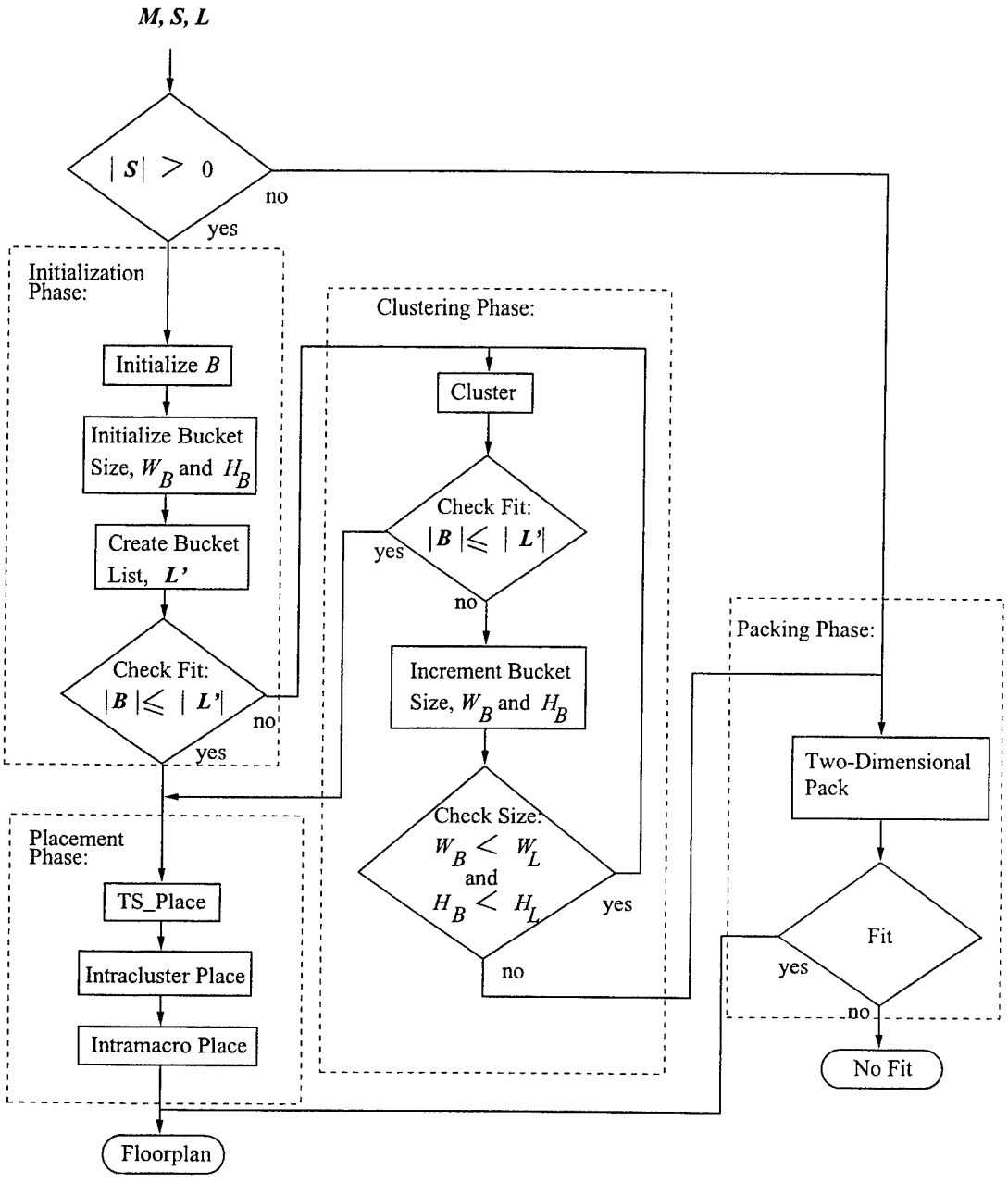


Figure 7.3: **Floorplanner execution flow**

```
Algorithm TS_FP(M,S,L)
  begin
    let success = TRUE;
    if | S | > 0 then
      (* initialize the buckets *)
      ∀ m_i ∈ M let b_i = {m_i};
      (* determine initial bucket size, H_B and W_B *)
      Create_Buckets(M);
      create L' where W_L' = ⌊W_L/W_B⌋ and H_L' = ⌊H_L/H_B⌋;
      success = checkfit(B,L');
      while(NOT success AND W_B < W_L AND H_B < H_L)
        B = cluster(M,S,H_B,W_B);
        success = checkfit(B,L');
        if NOT success then
          increment bucket size (H_B and/or W_B);
          update L' so W_L' = ⌊W_L/W_B⌋ and H_L' = ⌊H_L/H_B⌋;
        end if;
      end while;
      if success then
        TS_place(B,S,L');
        ∀ b_i ∈ B{
          intracluster_place(b_i,H_B,W_B);
          ∀ m_j ∈ b_i intramacro_place(m_j,b_i,H_B,W_B);
        }
      end if;
    if | S | = 0 OR NOT success then
      success = pack(M,L);
    end if;
    if NOT success then
      return "ERROR: circuit not floorplanned";
    end if;
  end;
```

two-dimensional array L.

In Figure 7.3, if $| S | = 0$ or if $| B | > | L' |$, we proceed to the two-dimensional packing (see subsection 7.3.8). In this phase we separate M into MH and MS. Then, we perform packing on MH. If MH fits the given area, we floorplan the soft macros in MS. If M fits, after this phase, every logic block making up the circuit or design netlist will have a physical location on the two-dimensional array L. Otherwise, the circuit does not currently fit the given area. The floorplanning process is summarized in Algorithm TS_FP(M,S,L).

In Figure 7.3, if the circuit or design netlist will not fit we have four options. We can reduce the size of the macro set M by partitioning the design spatially or temporally. We can increase the size of the target two-dimensional array L. This assumes a larger FPGA part is available. We can flatten the netlist and attempt to use standard placement techniques. This will become more difficult as design sizes get larger. Finally, if possible we can soften some of the hard macros to allow better space utilization.

7.3.1 Initializing Bucket Size

In this subsection, we describe the method for determining the initial bucket size which subsequently defines $| L' |$. The main goal of our floorplanning method was fast execution time. Therefore, we quickly initialize the width of the bucket, W_B, to the width of the widest macro cell

(hard or soft[8]). Similarly we initialize the height of the bucket, H_B, to the height of the tallest macro cell (hard or soft). This guarantees that any macro $m_i \in M$ will fit in any bucket. The procedure used to determine the initial W_B and H_B is shown in procedure Create_Buckets(M). In procedure Create_Buckets(M), $H(m_i)$ returns the height of macro m_i and $W(m_i)$ returns the width of macro m_i.

7.3.2 Bucket List, L'

The set of buckets[9], L', is created by dividing the set L into rectangles of equal size. The width of L' (in number of buckets) is defined as $W_{L'} = \lfloor \frac{W_L}{W_B} \rfloor$ and the height of L' (in number of buckets) is defined as $H_{L'} = \lfloor \frac{H_L}{H_B} \rfloor$. (Note, W_L and H_L define the width and the height (in number of logic blocks) respectively of the two-dimensional array L.) Therefore, $\mid L' \mid = H_{L'} \times W_{L'}$. Figure 7.4 shows an example L' for a 7 logic block $\times$ 6 logic block L ($W_L = 6$ and $H_L = 7$) and a 6 logic block $\times$ 2 logic block bucket ($W_B = 2$ and $H_B = 6$).

7.3.3 Clustering

As stated earlier, the set B is created or initialized by assigning each $m_i \in M$ to $b_i \in B$, and initially, $\mid B \mid = \mid M \mid$. When necessary, the size of set B is reduced by clustering elements of M so more than one element of M is in some $b_i \in B$. There is no limit placed on the maximum number of macros in each b_i as long as size constraints are satisfied. Size restrictions (described below) limit the macros used to form each cluster, $b_i \in B$.

Each cluster $b_i \in B$ is divided into two parts, a hard macro part and a soft macro part. The size restriction on b_i requires the total area of the hard macro part plus the total area of the soft macro part be less than or equal to the size of the bucket ($H_B \times W_B$). We define the width of the hard macro part of each cluster b_i as the sum of the width of the hard macros in b_i,

$$HMW(b_i) = \sum_{\forall m_j \in b_i \mid m_j \text{ is hard}} W(m_j) \ , \tag{7.1}$$

where $W(m_j)$ is the width of macro m_j in cluster b_i. We define the area for the hard macro part for each cluster b_i as the width of the hard macro part times the height of the bucket

$$HMA(b_i) = HMW(b_i) \times H_B \ . \tag{7.2}$$

The size for the soft macro part for b_i is defined as the width of the bucket minus the width of the hard macro part times the height of the bucket

$$SMA(b_i) = (W_B - HMW(b_i)) \times H_B \ . \tag{7.3}$$

The sum of the areas of all soft macros in b_i must be less than or equal to $SMA(b_i)$.

With these area constraints in mind, the set M is clustered to form the set B. The clustering method is derived from the connectivity work done in [87]. The connectivity cost function

[8]This assumes soft macros are supplied with some initial shape. Effort is made to maintain the shape of soft macros. The shape of soft macros is only changed if required to make the circuit fit the given area.

[9]a bucket is a set or group of physical logic block locations from L such that each bucket has the same size and shape ($H_B \times W_B$)

```
Procedure Create_Buckets(M):
  begin
    initialize W_B = H_B = 0;
    for i = 1 to | M |
      if W_B < W(m_i) then
        W_B = W(m_i);
      end if;
      if H_B < H(m_i) then
        H_B = H(m_i);
      end if;
    end for;
    return W_B and H_B;
  end;
```

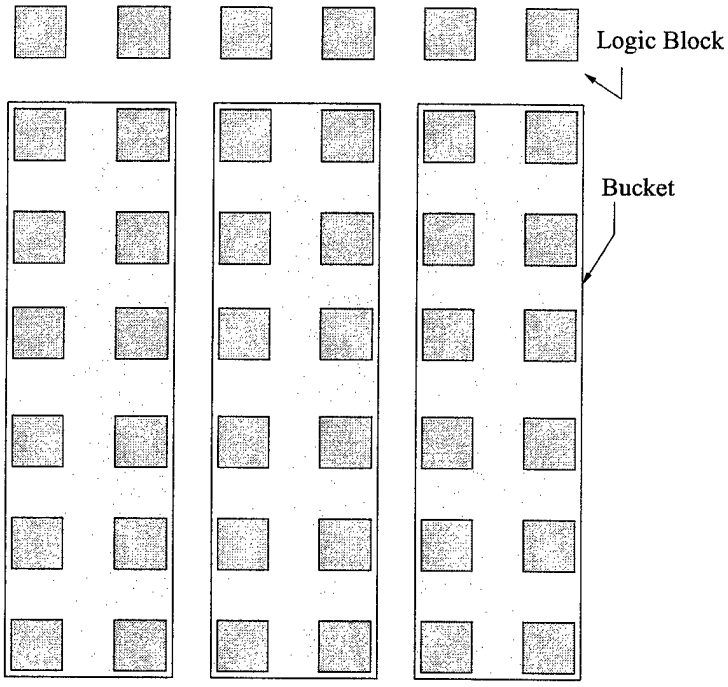


Figure 7.4: **Example L' made up of three 6×2 buckets**

Procedure Cluster(M,S,H_B,W_B,L'):
  begin
    $\forall\ m_i \in M$ let $b_i = \{m_i\}$;
    calculate $c_{ij}\ \forall\ b_i$ and $b_j \in B$;
    while $\mid B \mid > \mid L' \mid$ AND $\exists c_{ij} > 0$
      choose m_i and m_j with highest connectivity, c_{ij};
      let $b_i = m_i \cup m_j$;
      let $b_j = \phi$;
      update connectivity between clusters;
    end while;
    return B;
  end;

includes area constraints. Our connectivity cost function is summarized below.

$$c_{ij} = feas(i,j) \cdot \sum_{s_k \in S_{m_i} \cap S_{m_j}} \frac{1}{(\mid s_k \mid -1)} \cdot \frac{A_{tot}}{a_i + a_j} \cdot \frac{min(a_i, a_j)}{max(a_i, a_j)} \tag{7.4}$$

where a_i and a_j are areas of macro m_i and m_j respectively, A_{tot} is the total area of all macros, $\mid s_k \mid$ is the number of pins on signal s_k which connects macros m_i and m_j, $S_{m_i} \cap S_{m_j}$ is the set of all signals that connect macros m_i and m_j, and $feas(i,j)$ returns the feasibility of clustering m_i and m_j under size constraints described above. $feas(i,j)$ returns a 1 if it is possible to combine m_i with m_j else it returns a 0.

The clustering algorithm combines clusters with the highest connectivity to form larger clusters. In order to enhance routability, once area constraints have been met (i.e. $\mid B \mid \leq \mid L' \mid$) the algorithm stops and returns the set B. The clustering algorithm is summarized in procedure Cluster(M,S,H_B,W_B,L').

After clustering is complete, it returns the set B. The empty elements of B are removed, and each $b_i \in B$ consists of a unique list of elements from M. Here uniqueness implies $b_i \cap b_j = \phi\ \forall\ b_i \wedge b_j \in B \mid i \neq j$.

7.3.4 Increment Bucket Size

In the event that the first pass of clustering does not lead to a valid solution, the bucket size is increased to allow more flexibility during clustering. This increases the complexity of intracluster placement but allows more macros to fit in the same area. For example, consider floorplanning the 5 macros described in Table 7.1 so they fit on an L with $W_L = H_L = 6$ and $\mid L \mid = 36$. For the set M, both W_B and H_B will be set to 3 since these values reflect the largest macro width and height respectively. Figure 7.5 shows the buckets on L. Therefore, L' will initially have 4 buckets and M will not fit since $\mid B \mid > \mid L' \mid$. However, by doubling the width of the bucket, we can cluster m_1 and m_2 into one cluster and m_3, m_4, and m_5 into a second cluster that will fit in L.

7.3.5 Cluster Placement

Once the circuit is guaranteed to fit ($\mid B \mid \leq \mid L' \mid$) then the clusters $b_i \in B$ are placed using a two-step tabu search[10] (TS) based two-dimensional placement algorithm [15]. The first step of the

[10]tabu search is a meta-heuristic approach to solving optimization problems that (when used properly) approaches near optimal solutions in a relatively short amount of time compared to non-deterministic *random move* based methods [24]. Unlike approaches like simulated annealing or genetic algorithms that rely on a good random choice,

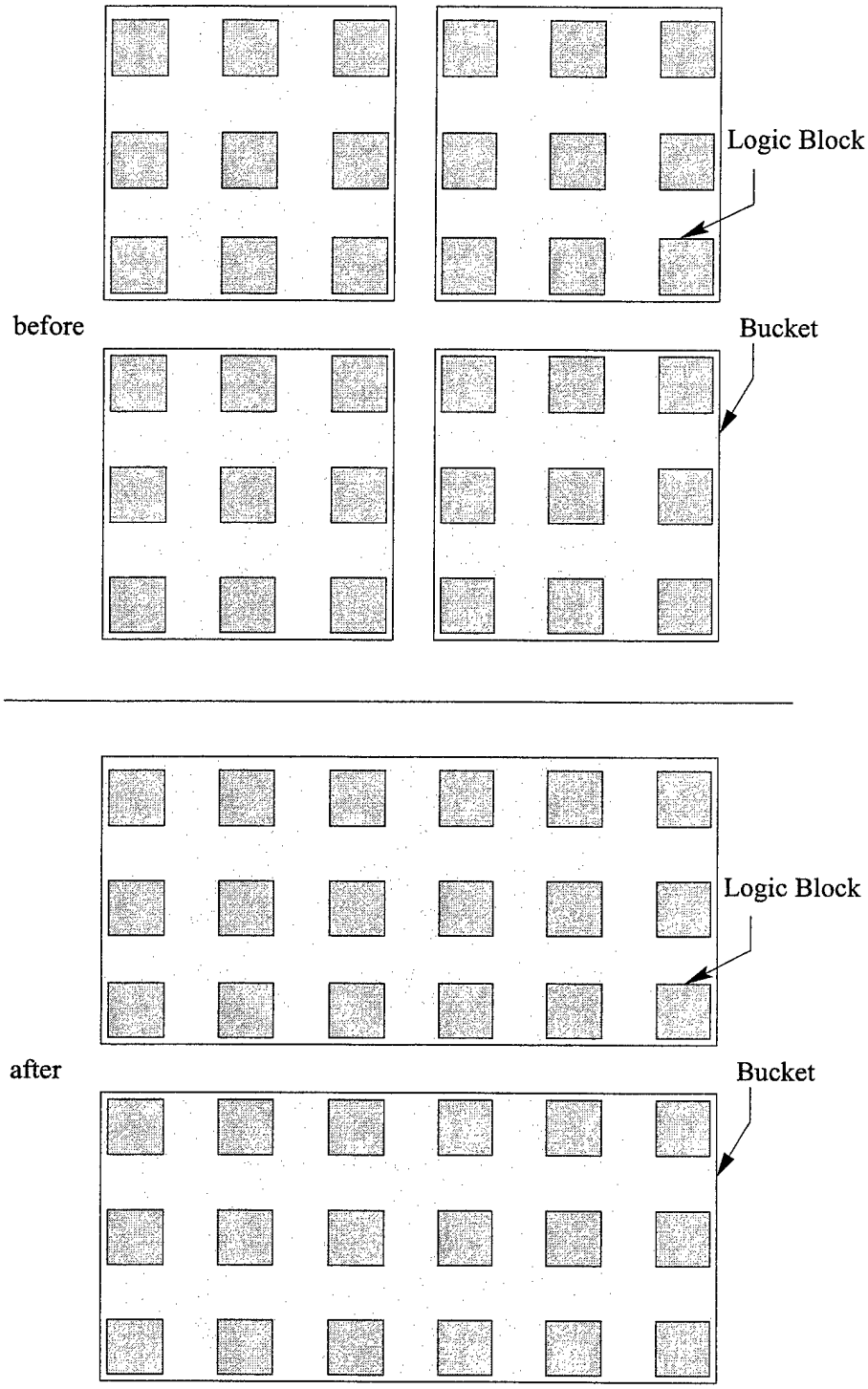


Figure 7.5: **Example L' made up of four 3×3 buckets converted to two 3×6 buckets**

Table 7.1: **Macro statistics for example floorplan**

Macro Statistics			
m_i	w_i	h_i	f_i
m_1	3	3	0
m_2	3	3	0
m_3	2	3	1
m_4	2	3	1
m_5	2	3	1

placement strategy minimizes the circuit's total wire length thereby enhancing the routability of the circuit. The second step attempts to average the circuit's edge lengths by weighting graph edges and minimizing the maximum weighted edge lengths.

For our TS approach, we convert each multi-terminal net to a set of edges where each edge consists of the driving terminal and one driven terminal. We use this model to keep net sources and sinks in close proximity thereby enhancing circuit performance. We create the set of edges by converting the hyper-graph input circuit model described earlier to a graph $G = (V, E)$ where $V = \{v_1, v_2, ...v_n\}$, $\mid V \mid = n$, $E = \{e_1, e_2, ...e_m\}$, and $\mid E \mid = m$. Each vertex $v_i \in V$ corresponds to a cluster $b_i \in B$ (if pad IO locations are available, we also include preplaced pseudo-elements of V representing the pad locations to help guide the placement). Each edge $e_i \in E$ connects a pair of vertices $(v_j, v_k) \mid v_j, v_k \in V$. The elements of E are created by considering each signal, $s_i \in S$. If we let m_{source} (where $m_{source} \in M_{s_i}$ and $m_{source} \in b_j$) be the source macro for signal s_i then an edge (v_j, v_k) is added to E for each sink on s_i such that $m_{sink} \in M_{s_i}$, $m_{sink} \in b_k$, and $j \neq k$. (In other words, an edge is added for each source/sink combination that are not in the same cluster.) At any given time, each element of V is mapped to a unique element of L', and the minimum requirement for mapping is $\mid V \mid \leq \mid L' \mid$.

The two-dimensional placement stage basically assigns each cluster to a unique bucket. After placement of each $b_i \in B$, each $b_i \in B$ will have associated with it a unique bucket $l'_j \in L'$. The physical location (on L) of each $b_i \in B$ in bucket l'_j can be found from the following equations:

$$X(b_i) = X(l'_j) \times W_B \tag{7.5}$$

and

$$Y(b_i) = Y(l'_j) \times H_B. \tag{7.6}$$

where $X(l'_j)$ returns the X-axis coordinate of l'_j on L' and $Y(l'_j)$ returns the Y-axis coordinate of l'_j on L'. After each cluster $b_i \in B$ is assigned a unique location on L, intracluster placement takes place to assign each $m_j \in b_i \in B$ a physical location on L. Intracluster placement also reshapes soft elements $m_k \in MS \in B$ that require further modification.

TS exploits both good and bad strategic choices to guide the search process. As a meta-heuristic, TS guides local heuristic search procedures beyond local optima. In TS, a list of possible moves is created. In the short term, as moves in the list are executed, *tabu*, or restrictions, are placed on the executed moves in order to avoid local optima. This *tabu* is typically in the form of a time limit, and unless certain conditions are met (e.g. *aspiration criteria*), the move will not be performed again until the time limit has expired.

7.3.6 Intracluster Placement

Once each cluster is assigned a location on L, the macros making up each cluster must be placed. Each macro $m_j \in M$ has associated with it a reference coordinate used to describe its physical location on the FPGA. Each logic block within each m_j also has a reference coordinate that describes its physical location relative to the reference coordinate for m_j. Intracluster placement is the task of assigning a reference coordinate from the set L to each macro $m_j \in b_i$, $\forall b_i \in B$, and, for any soft macro in M whose shape has changed, the task of assigning a set of reference coordinates for the logic blocks within the soft macro[11].

Intracluster or intrabucket placement for each $b_i \in B$ takes place in three steps. First, we place all hard macros by assigning each one an X,Y reference coordinate corresponding to some $l_j \in L$. Second, we place all soft macros by assigning each one an X,Y reference coordinate from L. Third, we change the shape of any soft macro that requires modification by assigning it a set of logic block coordinates relative to the reference coordinate of the soft macro. Figure 7.6 shows an example set of macros to be placed in the 9×12 Bucket 6 located at coordinates $X = 12$ and $Y = 18$. In Figure 7.6 each hard macro is labeled with $f = 0$ and each soft macro is labeled with $f = 1$. In this subsection we will describe each of the steps for intracluster placement.

Our feasibility check during clustering guarantees the hard macros in each b_i will fit by ordering them in the horizontal direction. Therefore for each $b_i \in B$, we place hard macros in a row, each with the same Y-axis coordinate. The Y-axis coordinate of each hard $m_j \in b_i$ is found from the following equation:

$$Y(m_j) = Y(b_i) \tag{7.7}$$

where $Y(b_i)$ returns the Y-axis coordinate (from the set L) of the bucket where cluster b_i was placed. To compute the X-axis coordinate of each hard $m_j \in b_i$ a sort key is computed for each hard $m_j \in b_i$ by averaging the X-axis coordinates of all $b_k \in B$ connected to m_j (this includes IO position information). Then the hard macros in b_i are reverse ordered according to the sort key and stored in an ordered list $\{q_1, q_2, ..., q_n\} = Q$. After ordering each hard macro in b_i, the X-axis coordinate of each hard macro in b_i is determined by the following. If we let q_k denote the kth element in the reverse ordered list of hard macros in b_i, then

$$X(q_k) = X(q_{k-1}) - W(q_k) \quad \forall \quad k > 1 \tag{7.8}$$

where $W(q_k)$ is the width of macro q_k, $X(q_{k-1})$ is the X-axis coordinate of macro q_{k-1}, and $X(q_1) = X(b_i) + B_W - W(q_1)$. For our example macros in Figure 7.6, since the Y-axis coordinate of the bucket is 18, the Y-axis coordinate for each hard macro(m_{16}, m_{19}, m_{27}, and m_{41}) is 18. If we assume the key for m_{16} is 3, m_{19} is 14, m_{27} is 13, and m_{41} is 43 then the X-axis coordinate for each hard macro is $X(m_{16}) = 16$, $X(m_{19}) = 20$, $X(m_{27}) = 18$, and $X(m_{41}) = 22$. Figure 7.7 shows the hard macros from Figure 7.6 placed in example Bucket 6.

We now describe the method for determining the X,Y reference coordinates for each soft macro. Similar to the method of ordering the list of hard macros for b_i, a sorting key is determined for each soft $m_j \in b_i$ by averaging the X-axis coordinates of all clusters connected to soft macro m_j (this includes IO position information). Then the soft macros in b_i are ordered according to the sort key and stored in an ordered list $\{r_1, r_2, \ldots, r_n\} = R$. After ordering each soft macro in b_i the X,Y reference coordinate of each soft macro in b_i is determined. If we let r_k denote the kth element in the ordered list of soft macros in b_i then the X-axis reference location of r_k is found

[11]Note: here only a set of reference coordinates is assigned for the set of logic blocks in the soft macro. The specific coordinates for each logic block in an altered soft macro are found during intramacro placement.

```
Procedure Find_Soft_X(b_i,r_k):
  begin
    if X(b_i) + X(r_{k-1}) is even then
      if lastY(r_{k-1}) ≠ Y(b_i) + H_B - 1 then
        X(r_k) = X(r_{k-1});
      else
        X(r_k) = X(r_{k-1}) + 1;
      end else if;
    else
      if lastY(r_{k-1}) ≠ 0 then
        X(r_k) = X(r_{k-1});
      else
        X(r_k) = X(r_{k-1}) + 1;
      end else if;
    end else if
  end;
```

from the procedure Find_Soft_X. In Find_Soft_X, $lastY(r_k)$ returns the Y-axis coordinate of the last element in macro r_k and $X(r_0) = X(b_i)$. If it is required that the soft macro r_k's shape be adjusted, then its Y-axis reference location is $Y(b_i)$, but if the soft macro's shape does not require adjustment, then r_k's Y-axis reference location is set relative to the Y-axis location of the last logic block in r_{k-1} ($lastY(r_{k-1})$). If we assume $r_1 = m_{21}$, $r_2 = m_7$, $r_3 = m_6$, and $r_4 = m_{13}$ for the soft macros in the example shown in Figure 7.6, then using the above methodology Figure 7.8 shows the final placement and shape for the macros assigned to example Bucket 6.

7.3.7 Intramacro Placement

After assigning the reference coordinates for hard and soft macros in each cluster, the logic blocks that make up any reshaped soft macro are placed using intramacro_place. Currently we use two methods for intramacro_place, and both are described below. Instead of actually performing full placement on the logic blocks within the soft macro, we incrementally reconfigure the placement of the logic blocks using a transform that matches the X and Y coordinates of the soft macro to the X and Y coordinates of the available space on L.

The first method for incrementally reconfiguring the placement is of $O(n)$ complexity, where n is the number of logic blocks within the reshaped soft macro. Starting from the leftmost-lowest coordinate of the soft macro, the logic blocks within the soft macro are matched to the leftmost-lowest coordinate available in the area of the bucket set aside for the soft macro. This methodology, though fast in execution, can substantially increase the length of nets connecting logic blocks; however, since the delay of the logic block is currently much greater than the interconnect delay, no substantial degradation to performance was noted.

The second method (designed to counter any performance degradation due to increased interconnect length) uses a minimax matching strategy to match locations of the logic blocks within the soft macro to coordinates available in the area of the bucket set aside for the soft macro. We use general minimax grid matching to accomplish this match.

7.3.8 Pack

In the event that no circuit netlist is available or the circuit will not fit using the above floorplanning strategy, two-dimensional packing is performed. Several methods of packing were investigated [40, 6, 8, 5, 46, 4]. A method similar to [4] was developed because the method in [4]

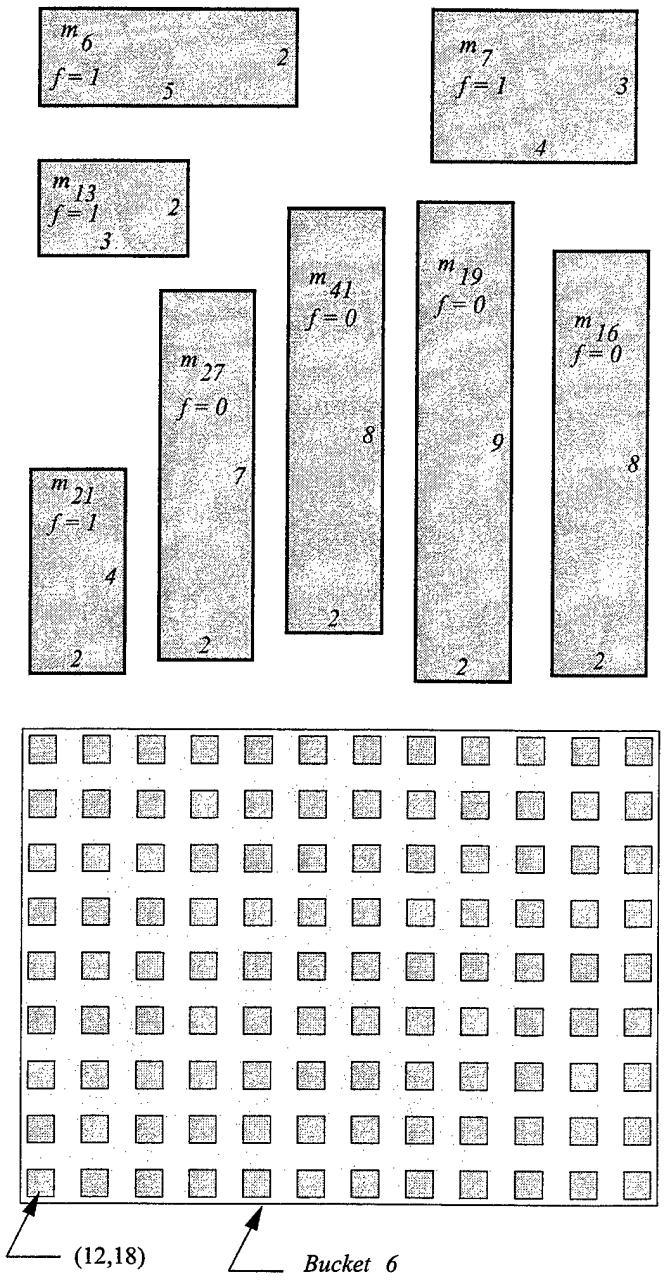


Figure 7.6: **Example set of hard and soft macros to be placed in Bucket 6 located at coordinate** (12,18)

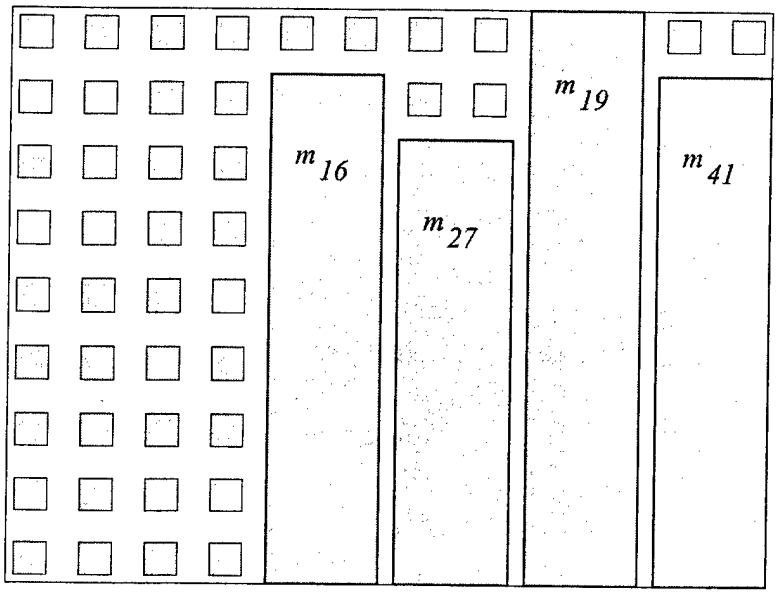


Figure 7.7: **Example hard macro placement for macros shown in previous figure**

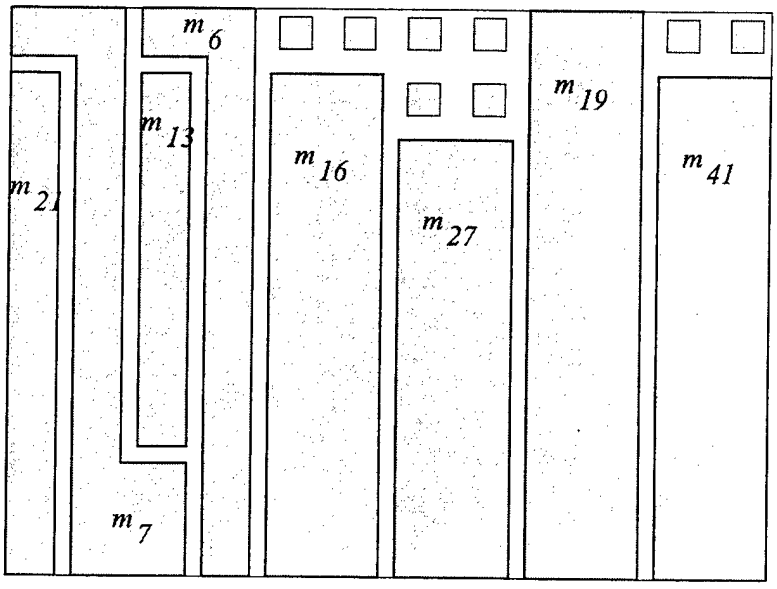


Figure 7.8: **Example placement of hard and soft macros**

```
Procedure pack(M,L)
  begin
    let success = TRUE;
    let MS = set of soft macros in M;
    let MH = M - MS;
    let W_S = soft_pack(MS,L);
    let W_H = hard_pack(MH,L);
    if W_S + W_H > W_L then
      success = FALSE;
    end if;
    return success;
  end;
```

```
Procedure pack_soft(MS,L)
  begin
    let W_S = infinity;
    intracluster_place(MS,H_L,W_S);
    ∀ m_j ∈ MS intramacro_place(m_j,MS,H_L,W_S);
    let W_S be the minimum width required for mapping MS;
    return W_S;
  end;
```

has a runtime complexity of $O(n \cdot log(n))$ where n is the number of modules and it bounds the width of final placement by $\frac{5}{4}\times$ the optimal value. Our method consists of a simple modification to the method presented in [4], and it is summarized below in procedure pack(M,L). First, we divide the set M into two sets MH and MS. Then, the elements of the set MS are floorplanned using procedure pack_soft(MS,L). The procedure pack_soft(MS,L) returns the width, W_S (in logic blocks), required to place (in a snake like pattern) the soft macros on the left hand side of L. It sets the coordinates and mapping for each logic block in each $m_i \in MS$. After pack_soft(MS,L), pack_hard(MH,L) is executed. It returns the width required for floorplanning the hard macros (W_H) and the x-axis and y-axis location for each hard macro. If the circuit fits, pack(M,L) returns success; otherwise, it returns unsuccessful.

The procedure pack_hard(MH,L) is used to perform two-dimensional packing on the set of hard macros MH. The method first divides the set MH into five regions: $R_i \mid i \in \{1,2,...5\}$. Each element $m_j \in MH$ is assigned to one of regions $R_{i=1}$ to $R_{i=4}$ if its height falls into one of the following ranges:

$$\left[\frac{1}{(i+1)}H_{max} \;,\; \frac{1}{i}H_{max}\right] \quad , \tag{7.9}$$

where $1 \leq i \leq 4$ and H_{max} is the height of the widest element of MH. Otherwise it is assigned to region R_5. The procedure pack_hard(MH,L) describes the procedure used to pack the set of hard macros.

In procedure pack_hard(MH,L), algorithm Left-Topmost, LT(R_i), packs pieces in the range $(1/(i+1)\ H_{max}, 1/i\ H_{max}]$. First, LT orders the elements in R_i by height, the highest elements are at the beginning of R_i. Then, LT packs pieces in order of non-increasing height by placing each piece as far left as possible in the region, and as high as possible at this leftmost position. In this way, the total height used decreases from left to right. This leaves a space increasing in height from left to right at the bottom of the region. The algorithm ROW uses this empty area of

```
Procedure pack_hard(MS,L)
  begin
    let W_H = infinity;
    let W = 1;
    for i = 1 to 4;
      LT(R_i);
      ROW(R_{i+1});
      NFDW(R_{i+1});
      let W_i = width of R_i;
      let W = W + W_i;
    end for;
    NFDW(R_5);
    let W_H be the width required for mapping MH;
    return W_H;
  end;
```

each region to pack in a row, pieces decreasing in height from right to left. When all pieces higher than $H_{max}/5$ have been packed, pieces of height at most $H_{max}/5$ are packed between LT packed pieces and the ROW packed pieces using the next-fit-decreasing-width (NFDW) algorithm. Also any pieces remaining in R_5 are packed using NFDW. Procedure NFDW packs pieces in irregular shaped regions. It packs pieces between widths labeled LEFTMOST and RIGHTMOST. According to [4], this method guarantees the packing to be within $5/4 \times$ the optimum width. Additionally, the complexity of the algorithm is $O(n \cdot log(n))$, where n is the number of macros in M [4].

7.4 Test Methodology

We empirically tested the floorplanning methodology described above using several macro based circuits (the circuits included both hard and soft macros). The top level macro based circuits were described using the `Xilinx Netlist Format` (XNF). The macros were also described using XNF files; however, they also included logic block placement information in the form of `RLOCs` so that all hard and soft macros were preplaced. The designs were mapped to the Xilinx XC4000E or XC4000XL family of FPGAs. Statistics for the macro based circuits are shown in Table 7.2.

For each circuit we obtained data for comparison in three ways. The first way we obtained data was to place and route flattened designs. We flattened each circuit netlist and removed all RLOC information. Then we used the Xilinx tools in the standard mapping approach (placement of logic blocks then routing of logic blocks) to map the circuit netlist. In the following tables, the results of this method are shown in columns labeled **Xilinx Flat**. The second way we obtained data for comparison was to floorplan and route the macro based circuits using the Xilinx tools. In the following tables, the results of this method are shown in columns labeled **Xilinx Macro**. The third way we obtained data was to floorplan the circuit with our TS_FP tool and route the circuits using the Xilinx tools. In the following tables, the results of this method are shown in columns labeled **TS_FP Macro**.

We used statistics available for the Xilinx tools to compare the three mapping methods. Specifically, we used static timing analysis available from Xilinx tools to compare the quality of the mapped circuits and report data from Xilinx tools to determine placement and routing times for Xilinx tools. Table 7.3 shows the tool used to place (flat designs only) or floorplan (macro based designs) each of the circuits as well as the Xilinx tool suite used for routing and static timing analysis. We used the `unix` *time* function to determine system floorplanning times for

Table 7.2: **Circuit statistics**

Macro Based Circuit Statistics						
Circuit Name	**Part**	$\mid M \mid$	**Total Area**	$\mid B \mid$	$\mid S \mid$	**Num IOs**
BOOTH	4013	64	264	72	473	33
CLA	4025	128	736	100	1024	133
CPU	4020	183	654	16	1051	38
MEDIAN	4013	39	295	15	392	80
MATMULT	4085	45	1998	35	891	306
BTCOMP	4036	97	403	81	768	264
XP-RI8	4025	31	723	12	417	170
XP-RI16	4085	18	2709	3	736	320
DCT	4085	122	3095	77	1089	113

Table 7.3: **Tools used for placing (flat netlist) or floorplanning (macro based netlist) test circuits. All circuit were routed using the corresponding Xilinx Router. All timing static timing analysis was performed on routed circuits**

Placement or Floorplanning Tools			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	PPR	PPR	TS_FP
CLA	PPR	PPR	TS_FP
CPU	PPR	PPR	TS_FP
MEDIAN	PPR	PPR	TS_FP
MATMULT	M1	M1	TS_FP
BTCOMP	M1	M1	TS_FP
XP-RI8	PPR	PPR	TS_FP
XP-RI16	M1	M1	TS_FP
DCT	M1	M1	TS_FP

TS_FP.

For the packing algorithm, we used the same macro based, benchmark circuits; however, we removed the interconnects and set $\mid S \mid$ to 0. Then, we floorplanned the circuits using only the two-dimensional packing procedure. After floorplanning, we added the interconnects to the floorplanned circuits and checked to see if the floorplanned circuits were routable.

An additional test was run using the floorplanning algorithm with a modified net length estimator. The modified net length estimator was based on work done at the University of California, Irvine by Min Xu [85]. The net length estimator takes advantage of the three line lengths available for routing in the Xilinx 4000 series of FPGAs (single, double, and long). It places a higher cost on routes that use more Programmable Interconnection Points (pips) (more pips are required to route nets that use multiple short segments instead of one longer segment). Used as a cost function to guide the floorplanner, this method adjusts the estimated cost of a net by looking at the Manhattan length of the horizontal and vertical components of the net, and assigning a cost that assumes the net will use the "best" length wiring resources when it is routed. Basically, each of the test circuits was floorplanned using this modified cost function.

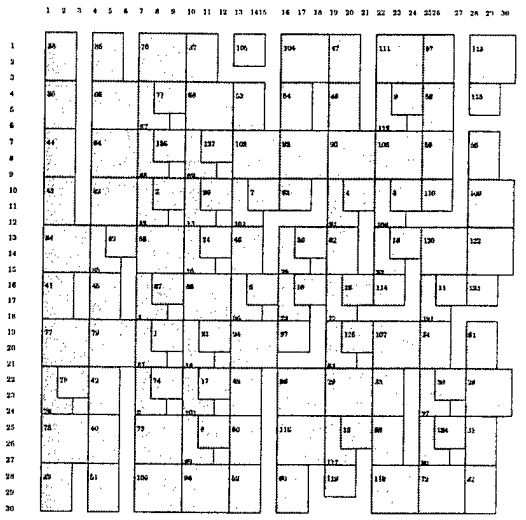


Figure 7.9: **Floorplan for CLA circuit**

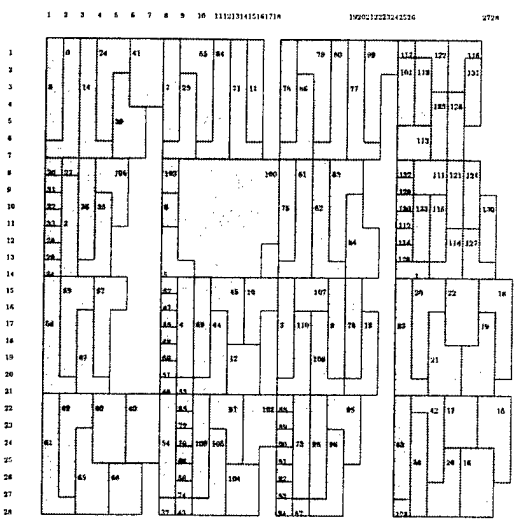


Figure 7.10: **Floorplan for CPU circuit**

7.5 Results and Analysis

Table 7.4 shows the execution times required to floorplan (or place in the case of the flattened netlists) the circuits. Column **TS_FP Macro** shows the execution times required by our methodology. Columns **Xilinx Flat**[12] and **Xilinx Macro**[13] show the execution times required by the `Xilinx` tools. Column **TS_FP Macro** shows a 45X improvement in execution time for our methodology over that of the commercial `Xilinx` tools. Table 7.4 also demonstrates execution speedup for working with macro based circuits versus flattened netlists. (It should be noted that the DCT design was not floorplanned using the `Xilinx` tools. On our `Sun Ultra 2`, we experienced memory faults during the circuit mapping process using the `M1` tool. For the same reason, we could not route or perform static timing analysis on the DCT design after floorplanning with our methodology; however, floorplanning execution time using our tool is shown.) All circuits (that did not cause memory faults) were 100% routable.

Table 7.5 shows the results of static timing analysis performed on the floorplanned circuits (Note: this data is taken from completely routed circuits). The values shown indicate the worst case pad to pad delay (in the case of combinational circuits) or the minimum allowable clock period (in the case of sequential circuits). From Table 7.5 we see the circuits floorplanned with our floorplanning methodology are similar in quality to those floorplanned by the commercial tools. Table 7.5 also shows there is not a substantial difference between delays encountered for our circuits with flat versus macro based netlists. This is probably due to the fact that logic block delay (for short distances or routes with few pips) is substantially greater than interconnect delay.

Table 7.6 gives the time taken for the Xilinx tools to route the circuits. This Table shows the time taken to route our floorplanned designs is similar to that of the Xilinx placed and routed designs. It should be noted that this time could be significantly reduced by using not just preplaced macros, but preplaced and prerouted macros. Figures 7.9, 7.10, 7.11, and 7.12 show example floorplans (from TS_FP) for the CLA, CPU, MATMULT, and DCT circuits respectively.

For the packing algorithm, we used the same macro based benchmark circuits. Not all packed circuits were routable, and of those that were routable, static timing analysis showed very poor

[12]Flattened netlist placed and routed by the `Xilinx` tools.

[13]Macro based netlist placed and routed by the `Xilinx` tools.

Table 7.4: **Floorplanning or placement execution times**

Execution times (cpu secs)			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	131	36	3.1
CLA	87	61	5.5
CPU	210	101	6.9
MEDIAN	70	34	1.3
MATMULT	876	634	3.2
BTCOMP	107	87	1.2
XP-RI8	315	83	1.2
XP-RI16	698	92	2.6
DCT	–	–	3.2

Table 7.5: **Floorplanned/placed circuit (post route) static timing analysis results**

Static Timing Analysis (ns)			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	49.5	46.8	50.0
CLA	97.2	105.1	124.4
CPU	95.6	106.9	103.3
MEDIAN	267.0	287.2	265.6
MATMULT	285.33	160.72	117.62
BTCOMP	124.09	150.74	127.58
XP-RI8	90.5	101.4	103.9
XP-RI16	296.86	283.70	289.21
DCT	–	–	–

Table 7.6: **Floorplanned/placed circuit routing times**

Routing Times (cpu secs)			
Circuit Name	**Xilinx Flat**	**Xilinx Macro**	**TS_FP Macro**
BOOTH	38	31	53
CLA	307	386	374
CPU	410	376	332
MEDIAN	30	116	44
MATMULT	358	271	295
BTCOMP	25	32	33
XP-RI8	552	776	1106
XP-RI16	192	507	596
DCT	–	–	–

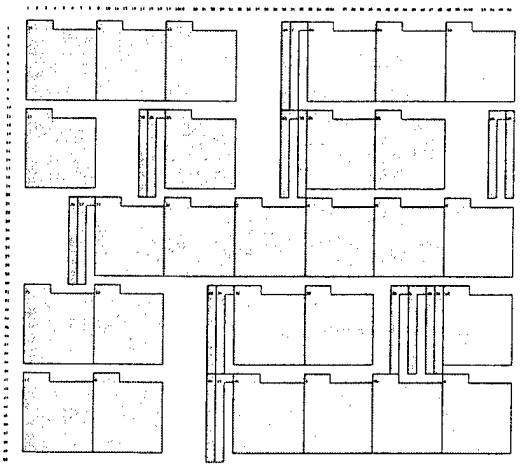


Figure 7.11: **Floorplan for MATMULT circuit**

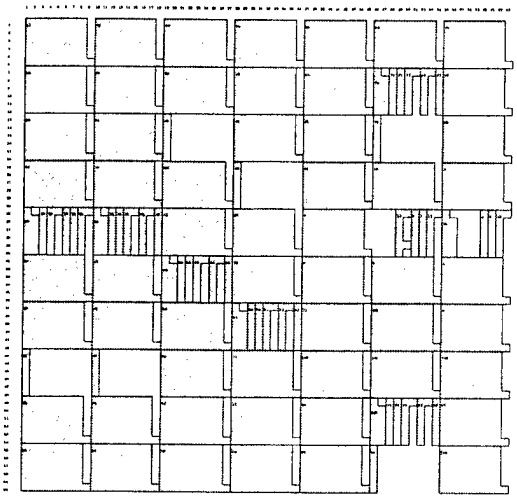


Figure 7.12: **Floorplan for DCT circuit**

performance. We did show the circuits could be floorplanned in the given area. An example circuit floorplanned using the pack algorithm is shown in Figure 7.13.

In this paragraph, we describe the results of using our TS based floorplanner with a cost function modified to reflect the work of Xu [85]. Overall, circuits floorplanned with this modified cost function were unroutable. Basically, the cost function placed a lower weight on long nets. This caused an unusually large number of nets to require "long" wiring resources for routing. Since these "long" wiring resources were quickly exhausted, the router had to revert to the "single" and "double" length wiring resources to route the nets. Due to the excessive total wire length in the mapped circuit, these resources were also quickly exhausted and the circuits were unroutable. One possible way to improve the use of this modified cost function is to limit the number of nets that can utilize the "long" lines and the "double" length lines available on the Xilinx architecture.

7.6 Conclusions

We have presented a performance driven fast floorplanning methodology for floorplanning macro based circuits. The methodology includes a clustering algorithm, placement algorithms, and a transform algorithm to quickly floorplan large macro based circuits. In the event that interconnect information is not available at the time of floorplanning our method uses two-dimensional packing to provide a preliminary floorplan. While flattening the netlist should provide better (relative to performance) results during the placement phase of the circuit, ever increasing circuit densities require an alternative method to handle large circuits in a timely (relative to execution time) fashion. Our approach shows dramatic improvement in the execution time without significant impact on quality of the mapped design.

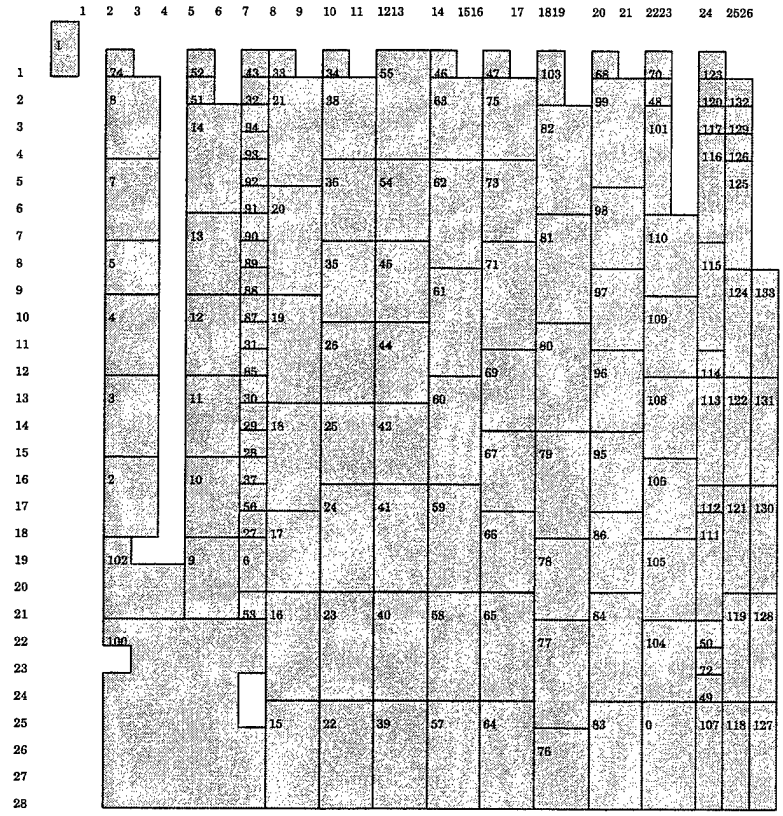


Figure 7.13: **Example circuit floorplanned using the** pack(M,L) *algorithm*

Chapter 8

Portable RC Development for Demonstration

A Portable Reconfigurable Computer Chasis called PARC has been developed for demonstration of field applications. Following are the key required components of PARC.

1. **Motherboard**
A PCI based motherboard for any processor which must be able to handle 1 full size PCI card, have builtin IDE support, be able to operate without a keyboard and be able to support ATAPI/IDE removable disk drives - the ZIP disk. Any chipset will work (VIA, SiS, Intel).

2. **Video Card**
Any video card will work. Recommend finding motherboard with built-in video cards. Common motherboards with built-in video use VIA or SiS chipsets which are acceptable.

3. **Memory**
Any memory of any size will work. A minimum of 32MB is recommended.

4. **ZIP Disk**
Must use an internal ZIP disk. Must be an ATAPI/IDE zip disk. There are different versions of ATAPI/IDE zip disk, recommended is the ATAPI version (for removable media recognition interface).There is an IDE only version, but that will not boot properly.

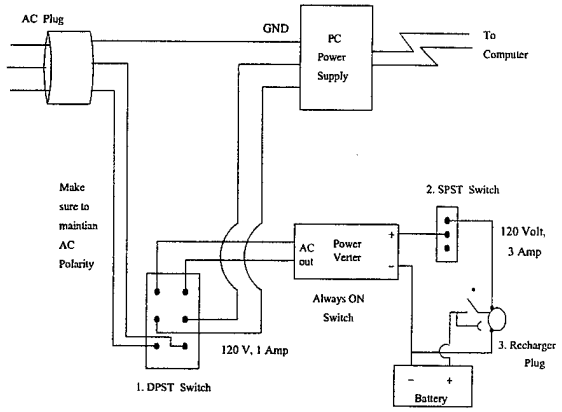
5. **Battery**
Need a 12 volt battery. Must have smallest dimensions to fit case. Must have highest mAh rating for size.Currently PARC has 3600 mAh battery (you will have to measure to get size). Estimated current draw without Wildforce board is 1200 mA per hour - 3 hour battery life. Estimated current with Wildforce is 1800-2000 mA per hour - 2 hour battery life.

6. **Case**
Need the cheapest case - those tend to be made with aluminum chassis. Smaller the case, the less room for battery. Aluminum chassis are lightest cases. Make sure chassis has room for battery.

7. **Powerverter**
Need a powerverter to create 120 volts from 12 volts. Recommend Tripplite PV300 which is an ultra compact powerverter that handles 300 watts and has best current handling capabilities. Figure 8.1 shows the Power circuit for PARC.

8. **FPGA Co-Processor**

Any user-provided PCI - compatible FPGA card can be accomadated. For demonstrations, an AMS Wildforce board has been used.



1. That DPDT switch is the Bottom View. Must be able to handle 120 Volts and at 1 Amp.

2. This switch can be SPST or DPDT but must handle 3 Amps atleast.

3. The Recharger Plug must have Disconnect feature.(see * in the figure)

Figure 8.1: **PARC Power Circuit**

Chapter 9

Prototype Software Development, Testing and Demonstration

9.1 Introduction

With the advancement of the field-programmable device technology, the Reconfigurable Computer (RC) that consists of a multi-FPGA board with memory banks and interconnection fabric, is being widely used for realizing fast implementation of wide classes of algorithms. Over the last couple of years there have been some research efforts [139, 140] towards design automation techniques for *dynamic reconfiguration* of the RC during the execution of a single application, leading to better performance and cost advantages. In order to achieve this, it is necessary to develop efficient partitioning and synthesis techniques that are independent of the RC architecture. In particular, the tasks of *Temporal Partitioning, Spatial Partitioning, and High-Level Synthesis* are central to the design process for dynamically reconfigurable architectures.

In the last decade, logic and layout synthesis techniques for FPGA s have matured greatly and commercial tools were developed to automate these tasks. However, high-level synthesis and multi-FPGA partitioning, both spatial and temporal, are still in their nascent stages and need to be further developed before commercial tools can appear.

The SPARCS system [140] (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) is a prototype design environment that incorporates efficient techniques for temporal and spatial partitioning and high-level synthesis. It provides a tightly integrated collection of partitioning and synthesis tools that help in automation of the design process for dynamically reconfigurable architectures. A brief summary of each major task in the SPARCS design flow is presented. For a more detailed discussion of SPARCS environment and its tasks, we refer the reader to [140].

In this chapter, we illustrate the steps involved in a typical partitioning and synthesis design flow for dynamically reconfigurable architectures, such as the SPARCS design flow. Using a real world design example, the Discrete Cosine Transform (DCT) subtask of the JPEG still image compression algorithm, this chapter aims to show that:

- A well-defined design flow, consisting of a tightly integrated collection of partitioning and synthesis tools, can provide a realistic design environment for dynamically reconfigurable architectures.

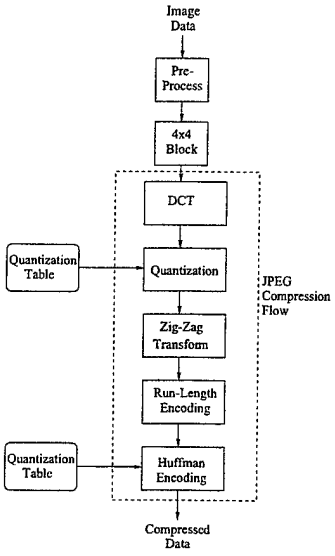


Figure 9.1: **JPEG Image Compression Standard**

- Dynamic reconfiguration does provide a performance/cost advantage over static configuration, for typical applications such as the JPEG that demand high-performance and inexpensive implementations.

The rest of this chapter is organized as follows: Section 9.2 explains the design example used for this chapter and the corresponding experimental setup. Section 9.3 provides an overview of the SPARCS design flow and related algorithms. It also briefly discusses the partitioning and synthesis of the design example. In Section 9.4, we present a detailed discussion of the results comparing the two versions of the JPEG algorithm that have the DCT subtask statically and dynamically reconfigured. Finally we make some concluding remarks in Section 9.5.

9.2 Design Example

The Joint Photographic Experts Group established the JPEG still image compression standard [141, 142]. The tasks of the JPEG compression standard are shown in Figure 9.1. The pre-processing stage partitions an image into 8x8 *blocks* of pixels, each of which pass through the following four major subtasks: The Discrete Cosine Transform (DCT), Quantization, Zig-Zag transformation, and Huffman encoding. We designed a JPEG-like still image compression algorithm [143] that works on 4x4 image blocks instead of the standard 8x8. Our implementation of the JPEG image compression algorithm, after extensive testing over several image files, achieved an average compression factor of 30.

We modeled our JPEG-like image compression algorithm as a *Hardware-Software Codesign* [143]. Software profiling of the JPEG compression algorithm revealed that DCT was the most computationally intensive subtask, consuming over 77% of the execution time. Therefore, we decided to implement DCT in hardware and the rest of the JPEG subtasks in software. The hardware that was used, is a Wildforce reconfigurable board consisting of two Xilinx XC4005 FPGAs having 196 CLBs each, and two 8K memory banks with a 16-bit word. The host computer that was used, is a Pentium PC with a 100MHz processor. The board can be plugged

into the backplane of the host computer through a PC Bus.

The FPGA-board follows a simple handshaking protocol with the host PC. The software that runs on the host initiates the communication by writing data on the board memory and then issues a signal for the board to start execution. When the board executes the design, the FPGAs can read/write data from/to their local memories. After the board finishes execution, the software running on the host PC reads back the resulting data values.

In order to study the partitioning and synthesis design flow through SPARCS, we considered the DCT subtask as our design example. The DCT for a 4x4 image block can be defined by Equation 9.1, given below. This equation has been derived from the definitions given in [141].

$$F(u,v) = \frac{1}{4}C(u)C(v)[\sum_{x=0}^{3}\sum_{y=0}^{3} f(x,y) * coeff(x,y,u,v)] \tag{9.1}$$

where $coeff(x,y,u,v) = \cos\frac{(2x+1)u\pi}{8}\cos\frac{(2y+1)v\pi}{8}$

The DCT is the first subtask of the JPEG compression algorithm. It processes one image block (4x4 pixels) at a time and produces a set of discrete cosine transform coefficients, one corresponding to every pixel entry in the 4x4 image block. Taking a closer look at Equation 9.1, we can simplify the discrete cosine transform by pre-computing the cosine coefficients and storing them in a 4x4 DCT coefficient matrix. Now the DCT can be viewed as two consecutive 4x4 matrix multiplications, as defined by Equation 9.2.

$$DCT = \frac{1}{4}\ C * (C * I)^T \tag{9.2}$$

Here, I is an input image block (4x4 pixel matrix) and C is the 4x4 matrix of DCT coefficients. A portion of the 4x4 DCT dataflow graph that performs two-vector (row * column) multiplications is shown in Figure 9.3. The following section presents a typical design flow through the SPARCS system using the DCT example as a case study.

9.3 The Design Flow

We present a typical design flow for dynamically reconfigurable architectures using the SPARCS system [140] and the DCT example as a case study. Section 9.3.1 briefly discusses the SPARCS system. Section 9.3.2 discusses the DCT Task Graph. Section 9.3.3 presents a typical ILP formulation for temporal partitioning. Section 9.3.4 presents a typical genetic algorithm based solution to the spatial partitioning problem. Finally, section 9.3.5 discusses issues related to high-level synthesis.

9.3.1 SPARCS System

In this section, we only present an overview of the SPARCS system. For more details, we refer readers to [140]. Figure 9.2 shows the design flow through SPARCS. The SPARCS system accepts a design specification at the behavior level, in the form of task graphs, specified in VHDL. The tasks may communicate to each other either directly (following a protocol) or through shared memories. In order to re-target the design to different FPGA boards, the SPARCS system takes,

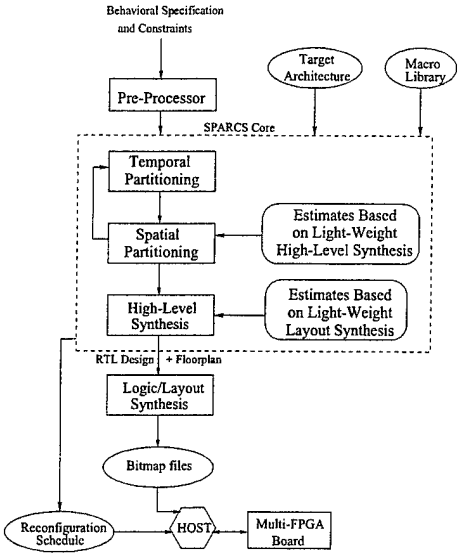


Figure 9.2: **The SPARCS Design Flow**

as an input, the target architecture specification to which the design has to be mapped. In addition, the system takes user constraints and requires a macro component library.

The SPARCS system contains a temporal partitioning tool to temporally divide and schedule the tasks on the reconfigurable architecture, a spatial partitioning tool to map the tasks to individual FPGAs, and a high-level synthesis tool to synthesize efficient register-transfer level designs for each set of tasks destined to be down-loaded on each FPGA. Following partitioning and synthesis, commercial logic and layout synthesis tools are used to generate bit-map files for each configuration of each FPGA. The SPARCS system also generates a reconfiguration program which can be used to control the RC's reconfiguration and execution from a host computer.

9.3.2 DCT Task Graph

As our first step, the behavioral specification of the 4x4 DCT was modeled in the form of 32 matrix equations (16 for each matrix multiplication) and simulated in VHDL. We then partitioned the entire DCT dataflow graph into a collection of tasks. The task partitions for a single DCT block is shown in Figure 9.3. There are 16 such blocks in the entire 4x4 DCT dataflow graph. The two inputs to each multiplication operation are: a constant DCT coefficient and either an input pixel (for tasks T1 and T2), or an intermediate output of an 18-bit addition (for tasks T3, T4, T6, and T7). Notice that while deciding on task boundaries, we ensured that each task does not exceed an area of 196 CLBs for the XC4005. This is because the SPARCS system requires that each task should individually fit on the FPGA. For example, task T1 consists of two 9-bit multiplications, a 17-bit addition and an 18-bit addition, for a total of 134 CLBs. Whereas tasks T4 and T6 contain one 16-bit multiplication which in itself occupies 127 CLBs. Table 9.1 shows the area and delay estimates for all operations in the graph, and Table 9.2 shows the summed up areas for each task. These value were obtained from the pre-characterized RTL component library that is used by SPARCS system. The task boundaries were also based on minimizing the amount of communication between the tasks. After drawing the task boundaries, the DCT task graph consisted of a total of 112 tasks.

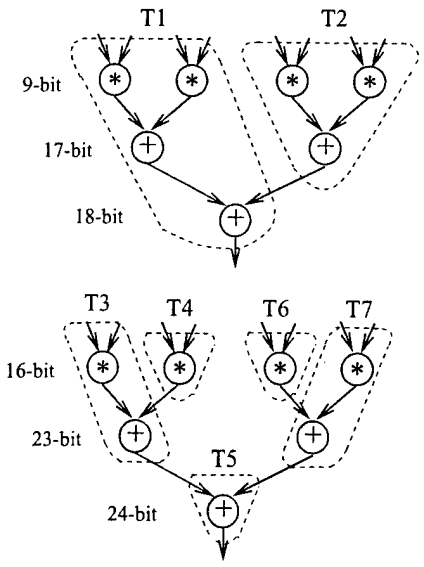


Figure 9.3: **DCT Graph and Task Partitions**

Table 9.1: **Estimates for DCT Operations**

Operation		Area	Delay
Type	Bits	(CLBs)	(ns)
*	9	42	68
+	18	25	80
*	16	127	115
+	24	34	86

Note that the tasks also contain memory read and write operations but these are not shown in the figure for the sake of clarity. Another goal while deciding the task boundaries was to minimize the number of memory read/write operations. Since the SPARCS design model allows direct communication between tasks, certain inter-task communications were not done through memory. For example, the output of task T2 is directly communicated to task T1, thereby reducing two memory cycles. This direct communication channel is viewed as a constraint by the SPARCS partitioning system. For example, the temporal partitioner is restricted to hold these tasks together in the same temporal segment.

Table 9.2: **Area Estimates for DCT Tasks**

Task	T1	T2	T3	T4	T5
Area	134	109	160	127	34

9.3.3 Temporal Partitioning

As mentioned in Section 9.3.1, the behavior specification is in the form of a task-level dependency graph. The control dependency represents the execution sequence of the tasks. If the execution of task B is control-dependent on task A, then A has to be executed in the same or earlier segment as B. The channel dependencies have edge weights that represent the amount of data to be stored and retrieved, if the two tasks connected by an edge are placed in different temporal segments.

The temporal partitioner has an abstract view of the underlying board resources and uses aggregate costs for temporal partitioning. From the RC architecture specification, the overall resource constraint (C) and shared memory size (M_s) are derived. For example, a typical resource constraint would be the total number of function generators (FGs) derived by adding the FGs of all the FPGAs on the board.

The temporal partitioner heuristically estimates the upper-bound on the number of temporal segments (N) for the Non-Linear Programming (NLP) formulation using a fast list-scheduling heuristic (a variation of [139]). As part of the formulation, we have incorporated a synthesis model to determine the resource sharing among tasks. This requires an operation level modeling of each task for the synthesis subproblem. The NLP model is linearized and solved by an ILP (Integer Linear Program) solver.

Terminology: $t_i \rightarrow t_j$ - directed edge between tasks, $t_i, t_j \in T$, representing a control or channel dependency; $i_i \rightarrow i_j$ - a directed edge between operations, $i_i, i_j \in I$ exists; $Bandwidth(t_i, t_j)$ - number of data units to be communicated between tasks t_i and t_j; $Op(t)$ - the set of all the operations in the operation graph of task t. The operation graph for a task are all the operations in the task with their data/control flow dependencies; $Fu(i)$ - the set of functional units on which operation i can execute; $CS(i)$ - the set of control steps over which operation i can be scheduled. $CS(i)$ ranges from $ASAP(i)$ *to* $ALAP(i) + L$, where L is the relaxation over the maximum ALAP for the schedule. $ASAP(i)$ and $ALAP(i)$ are the As Soon As Possible and As Late As Possible control steps for operation i, which are derived by scheduling operations on unlimited resources; $CS^{-1}(j)$ - the set of operations which can be scheduled on control step j; F - the set of functional units corresponding to the most parallel schedule obtained from the high-level synthesis estimator; N - the upper bound on the number of temporal segments. The segments are numbered 1 *to* N, the index of the segment specifies the order of execution of the segments. Note that the generated optimal solution may have fewer than N segments; M_s - the shared memory available for storage between temporal segments; $FG(k)$ - the number of function-generators used for functional unit k; C - the resource capacity of the board; I - the set of all operations in the specification.

We assume for the current model that the latency of each functional unit is one control step, and the result of an operation is available at the end of the control step.

Non-Linear 0-1 Model: In this section, we describe the variables, constraints, and cost function used in the formulation of our non-linear programming model.

1. *Variables:* We have four sets of decision variables: y_{tp} models the partitioning at the task level, x_{ijk} models the synthesis subproblem at the operation level, $w_{pt_1t_2}$ models the communication cost incurred if two tasks connected to each other are not placed in the same segment and u_{pk} determines whether a functional unit has been used in a segment (F, obtained initially, is an upper-bound on the number of functional units that can be used in

a temporal segment.) All are 0-1 variables.

$$y_{tp} = \begin{cases} 1 & \text{if task } t \in T \text{ is placed in segment p,} \\ & 1 \leq p \leq \text{N} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ijk} = \begin{cases} 1 & \text{if operation } i \in I \text{ is placed in} \\ & \text{control step } j \in CS(i) \text{ and uses} \\ & \text{functional unit } k \in Fu(i) \\ 0 & \text{otherwise} \end{cases}$$

$$w_{pt_1t_2} = \begin{cases} 1 & \text{if task } t_1 \text{ is placed in any segment} \\ & 1 \cdots p-1 \text{ and } t_2 \text{ is placed in any} \\ & \text{of } p \cdots N \text{ and } t_1 \rightarrow t_2 \\ 0 & \text{otherwise} \end{cases}$$

$$u_{pk} = \begin{cases} 1 & \text{if functional unit } k \in F \text{ is used in} \\ & \text{segment p, } 1 \leq p \leq N \\ 0 & \text{otherwise} \end{cases}$$

y_{tp} and x_{ijk} are the fundamental modeling variables. All other variables are secondary and are non-linearly constrained in terms of the fundamental variables.

2. *Constraints* Temporal partitioning and synthesis has the following constraints:

Uniqueness Constraint: Each task should be placed in exactly one segment among the N temporal segments.

$$\forall t \in T \quad : \sum_{p=1}^{N} y_{tp} = 1 \tag{9.3}$$

Temporal order Constraint: A task t_1 on which another task t_2 is dependent cannot be placed in a later segment than the segment in which task t_2 is placed.

$$\forall t_2, \ \forall t_1 \rightarrow t_2, \ \forall p_2, \ 1 \leq p_2 \leq N-1 \quad :$$

$$\sum_{p_2 < p_1 \leq N} y_{t_1p_1} + y_{t_2p_2} \leq 1 \tag{9.4}$$

Shared Memory Constraint: The amount of intermediate data stored between segments should be less than the shared memory M_s. $w_{pt_1t_2}$, if 1, signifies that t_1 and t_2 have a data dependency and are being placed across temporal segment p. Therefore, the data being communicated between them, $Bandwidth(t_1, t_2)$, will have to be stored in the memory of segment p. The sum of all the data being communicated across a segment should be less than the available shared memory.

$$\forall p, \ 2 \leq p \leq N \quad :$$

$$\sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} (w_{pt_1t_2} * Bandwidth(t_1, t_2)) \leq M_s \tag{9.5}$$

Unique Operation Assignment Constraint: Each operation should be scheduled at one control step and on only one functional unit.

$$\forall i \quad : \sum_{k \in Fu(i)} \sum_{j \in CS(i)} x_{ijk} = 1 \tag{9.6}$$

Temporal Mapping Constraint: This constraint prevents more than one operation from being scheduled at the same control step on the same functional unit.

$$\forall j \quad : \sum_{k \in Fu(i)} \sum_{i \in CS^{-1}(j)} x_{ijk} \leq 1 \tag{9.7}$$

Dependency Constraint: To maintain the dependency relationship between operations, an operation i_1, whose output is necessary for operation i_2, should not be assigned a later control step than the control step to which i_2 is assigned.

$$\forall i_1 \rightarrow i_2, \ \forall j_2 \leq j_1, \ j_1 \in CS(i_1), \ j_2 \in CS(i_2) \quad :$$

$$\sum_{k_1 \in Fu(i_1)} x_{i_1 j_1 k_1} + \sum_{k_2 \in Fu(i_2)} x_{i_2 j_2 k_2} \leq 1 \tag{9.8}$$

Resource Constraint: Resource constraints are in terms of variables u_{pk}. Typical FPGA resources include function generators, combinational logic blocks (CLB) etc. Similar equations can be added if multiple resource types exist in the FPGAs. α is a user defined logic-optimization factor in the range [0;1]. Typical values [147] of α using Synopsis FPGA components are in the range [0.6;0.8].

$$\forall p, 1 \leq p \leq N \quad : \alpha * \sum_{k \in F} (u_{pk} * FG(k)) \leq C \tag{9.9}$$

Unique Control Step Constraint: Each control step is mapped uniquely to a temporal segment.

$$\forall t_1, \ \forall t_2 \neq t_1, \ \forall i_1 \in Op(t_1), \ \forall i_2 \in Op(t_2),$$

$$\forall j \in CS(i_1) \cap CS(i_2), \ \forall p_1, 1 \leq p_1 \leq N,$$

$$\forall p_2 \neq p_1, 1 \leq p_2 \leq N \quad :$$

$$\sum_{k_1 \in Fu(i_1)} x_{i_1 j k_1} * y_{t_1 p_1} + \sum_{k_2 \in Fu(i_2)} x_{i_2 j k_2} * y_{t_2 p_2} \leq 1 \tag{9.10}$$

3. *Cost Function:* Minimize the cost of data transfer between temporal segments. Minimize:

$$\sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} \sum_{1 \leq p \leq N} (w_{pt_1 t_2} * Bandwidth(t_1, t_2)) \tag{9.11}$$

We have presented only a part of the NLP model here. For more details about other constraints, linearization, and solution by ILP techniques refer to [144]. To reduce the amount of time required for solving the ILP model, the model may be solved to find a constraint satisfying solution rather than an optimal one. This leads to significant speedup in solving the ILP model.

For the DCT example, the temporal partitioning system will try to minimize the cost of data transfer. Consider the temporal partitioning of tasks T1, T2 and T3 shown in Figure 9.3. Under the given area constraint (196 CLBs per FPGA), all three tasks cannot fit together on a single temporal segment. Since there is a direct data transfer between task T1 and T2 the partitioner will place these tasks in the same temporal segment rather than placing tasks T1 & T3 or T2 & T3 together. For the same reason, the temporal partitioner will place tasks T3 and T4 in one temporal segment; this will minimize the data transfer between temporal segments. Given the DCT task graph as an input, the SPARCS temporal partitioner produced fifty temporal configurations as shown in Figure 9.4.

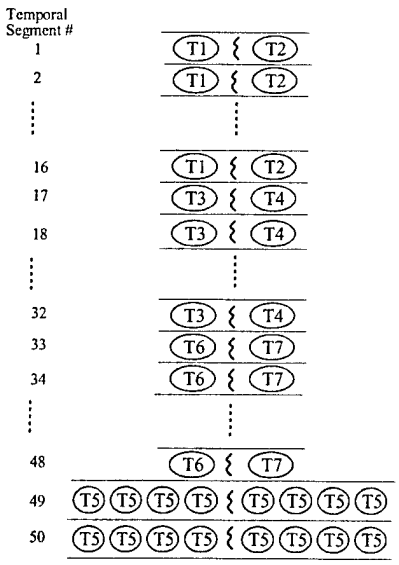


Figure 9.4: **Temporal Partitions for DCT**

9.3.4 Spatial Partitioning

Problem Formulation:

Let $\mathcal{F} = \{f_1, f_2, \cdots f_N\}$ be the N FPGAs available on the target reconfigurable board. Each FPGA has a set of attributes associated with it. For any $f \in \mathcal{F}$:

- $C(f)$ = number of function generators in f,
- $F(f)$ = number of flip-flops in f,
- $P(f)$ = number of uncommitted I/O pins in f,
- $L(f)$ = size of the local memory of f.

CM represents the direct connection matrix. It defines the number of dedicated lines pre-routed between each pair of FPGAs. I_c denotes the number of programmable interconnection channels available on the board.

A *spatial partition* of a task graph, $TG = (V, M, E)$, where V is the set of task nodes, M is the set of memory segments, and E is the set of dependency edges and channels, is a binding of each task in V to a unique FPGA and each logical memory segment to a unique local/shared memory, such that all architectural constraints are satisfied. These constraints are satisfied based on performance estimates obtained from a light-weight high-level synthesis estimator. When multiple valid spatial partitions exist, the one which produces the fastest implementation is chosen.

Spatial Partitioning Algorithm:

We model and solve the spatial partitioning problem through a *Genetic Algorithm* (GA). The genetic search procedure was developed by John Holland in 1975 [31], and since then has been used successfully for solving several combinatorial problems in VLSI design automation [148, 149, 61]. A genetic algorithm consists of an iterative procedure during which a series of *generations* of *populations*, one per iteration, are created. Each member of a population, also called chromosome, represents a solution of the problem being solved. The solution representation is based on a suitable encoding of the solution space.

From the minimization perspective, genetic algorithms attempt to discover an optimal – least cost – solution to the problem. The cost of a chromosome is evaluated by the partition performance and cost estimator which are discussed later. The GA uses an *evolution function* to generate a new generation p_{i+1} from an existing generation p_i. The evolution function usually consists of three components, called *operators*: *Selection, Crossover* and *Mutation.*

Following the generation of the new population, the current population is discarded and the new population becomes current. This evolution process continues until termination condition has been reached. The termination condition is either a constraint satisfying solution or an upper limit of the number of generations that GA explores.

Genetic Modeling for Spatial Partitioning:

Encoding: The solution representation must capture the binding of tasks to the FPGAs and the binding of logical memory segments to local/shared physical memories. We use a simple integer array to encode the above information. Each chromosome has two integer arrays - *task array* TA and *memory array* MA. The length of the TA is equal to the number of tasks in the task graph (t) and the length of the MA is equal to the number of memory segments (m). Consider a board having N FPGAs with local memories and a shared memory. For $1 \leq i \leq t$, the variable $TA[i]$, ranging from 1 through N, represents the FPGA number to which task i is assigned. Similarly, for $1 \leq i \leq m$, the variable $MA[i]$, ranging from 0 through N, represents the memory bindings. $MA[i] = 0$ implies that the memory segment i is mapped to the shared memory.

Initial Population: The task arrays for all chromosomes in the initial population are set to random legal values. Then based on the task assignments, for each chromosome, we assign the logical memory segments to local physical memories. If the majority of the tasks which access a memory segment are assigned to FPGA k then we bind the memory segment to the local memory of FPGA k.

Crossover: We use a uniform crossover operator. A binary string, T, whose length is equal to the greater of the number of tasks and the number of memory segments, is generated. Each bit in this *template* is randomly set to either 0 or 1. Next, two parents are probabilistically selected for mating. Let pt_1, pt_2 be the task arrays and pm_1, pm_2 be the memory arrays in the parents. Then ct_1, ct_2, cm_1, and cm_2, are the corresponding arrays in the two child chromosomes resulting from a crossover that is defined as:

$$ct_1[i] = \begin{cases} pt_1[i] & \text{if } T(i) = 0 \\ pt_2[i] & \text{otherwise} \end{cases} \tag{9.12}$$

$$ct_2[i] = \begin{cases} pt_1[i] & \text{if } T(i) = 1 \\ pt_2[i] & \text{otherwise} \end{cases} \tag{9.13}$$

$$cm_1[j] = \begin{cases} pm_1[j] & \text{if } T(j) = 0 \\ pm_2[j] & \text{otherwise} \end{cases} \tag{9.14}$$

$$cm_2[j] = \begin{cases} pm_1[j] & \text{if } T(j) = 1 \\ pm_2[j] & \text{otherwise} \end{cases} \tag{9.15}$$

In the above equations, i and j have legal values based on the number of tasks and memory segments in the task graph.

Mutation: The mutation operator randomly selects an entry from the chromosome arrays and changes its value to another legal value. Effectively, the mapping of a single task or a memory segment is modified.

Partition Cost Estimation:

The cost of each chromosome (spatial partition) is dependent on several constraint satisfaction requirements:

- **Area Constraint (A) and Speed Constraint (S)**: The spatial partitioner invokes high-level synthesis estimation routines (light-weight HLS) to gather area estimates and verify speed constraint satisfaction. The HLS routines report all area and speed constraint violations.
- **Pin Constraints (P)**: The spatial partitioner ensures that there are enough pins available on the FPGAs to perform inter-FPGA and FPGA -memory communications.
- **Interconnect Constraint (I)**: The reconfigurable board usually has a limited number of programmable channels to interconnect the FPGA and memories. There may also be dedicated lines between FPGAs.
- **Memory Constraint (M)**: Logical memory assignments must not violate the memory bandwidth requirements on all physical memories at any time during the execution of the design.

Let $\Delta A, \Delta S, \Delta P, \Delta I$, and ΔM be the respective constraint violation values for a given chromosome c. In the case when any constraint is met, the Δ value is zero. For example, if all area constraints are met then ΔA is zero. The cost of the chromosome c is given by:

$$cost(c) = \frac{\Delta A}{A} + \frac{\Delta S}{S} + \frac{\Delta P}{P} + \frac{\Delta I}{I} + \frac{\Delta M}{M} \tag{9.16}$$

The above form of cost function is widely used in several domains where a set of conflicting constraints are to be met [61, 150, 80]. In the case when the spatial partitioner cannot achieve a constraint satisfying solution, it flags a *failure* and returns tighter constraints for use by the temporal partitioner. The new aggregate constraints are based on the degree of cost violated by the best achieved partition.

For the case study example, the spatial partitioner will use area estimates of the DCT tasks listed in Table 9.2. The spatial partitioner uses the HLS tool as an estimator to get estimates on a spatial partition (collection of tasks) that it is contemplating for a single FPGA. For all tasks except T5, the areas are such that no two tasks can fit into the same FPGA. So the job of the spatial partitioner is simple, as can been seen in the temporal segments 1 through 48, shown in Figure 9.4. However, since task T5 is small, the spatial partitioner fitted four tasks in each FPGA as shown in the temporal segment numbers 49 and 50. While doing this, the spatial partitioner will consider the memory constraints among these tasks. The goal will be to join those tasks that access data from the same local memory.

9.3.5 High-Level Synthesis

The behavioral description (corresponding to a collection of tasks specified by the spatial partitioner) along with a pre-characterized macro library and the user constraints are taken

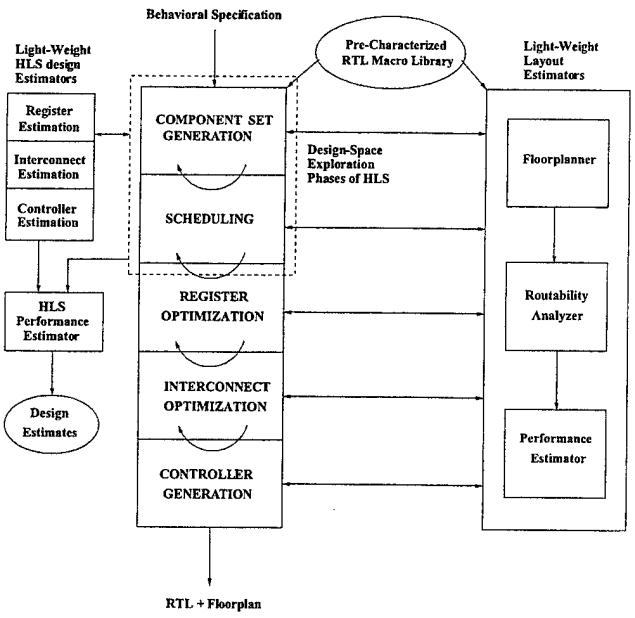


Figure 9.5: **Layout Integrated High-Level Synthesis**

through High-Level Synthesis (HLS) to obtain an equivalent Register-Transfer Level (RTL) implementation that can fit into a single FPGA chip. The RTL design [126, 152] consists of a *Datapath* which is a netlist of components picked from the component library, and a *Controller* which is a finite state machine that sequences the datapath components to perform the computations specified in the behavior of the design.

In the Layout-Integrated HLS design process shown in Figure 9.5, a collection of light-weight layout algorithms are integrated into HLS. The first is a floorplanner that picks the macros (from the pre-characterized macro library) and tries to place them on the FPGA , simultaneously trying to reduce the overall area and delay of the design. The second is a routability analyzer that checks the feasibility of routing after completing the placement. Finally, the performance estimator predicts the area and delay of the generated floorplan. The RTL design and the floorplan are then *predictably* taken through necessary phases of logic and layout synthesis to obtain the bit-stream for the FPGA configuration. Therefore, the estimates made during HLS are preserved and the FPGA implementation definitely satisfies the required constraints. There is another on-going work [151] that follows a similar approach.

At the core of the SPARCS system is a high-level synthesis tool, Asserta, which accepts behavioral descriptions specified in VHDL (as a collection of processes/tasks) and performance constraints in terms of the desired clock width and the upper limit on the area. The Asserta tool has been tailored to suit the layout-integrated synthesis approach for the SPARCS system, as shown in Figure 9.5. Asserta satisfies the clock width constraint by trying to minimize the maximum combinational delay of any register transfer. The area constraint is satisfied by trying to minimize the size of both the datapath and the controller. The HLS process consists of component set generation, scheduling and performance estimation, register and interconnect optimization, and controller generation. For a detailed discussion of these phases, we refer the reader to Roy et al. [32].

Table 9.3: **Area and Delay Estimates for DCT tasks**

DCT Task	Area (CLBs)	Clk Period (NanoSecs)	Number of C-steps
T1	186	177	5
T2	159	155	4
T3	168	163	4
T4	180	210	2
T5	45	150	6

For the case study, the tasks in the partitioned DCT, shown in Figure 9.4, were taken through Asserta to generate the RTL designs. The RTL designs were then taken through logic synthesis (Synopsys FPGA compiler) and layout synthesis (Xilinx M1 tools) to produce the FPGA configuration files. For the tasks T1 through T5, Table 9.3 shows the area estimates provided by the Xilinx PAR (Partitioning And Routing) tool, timing estimates provided by the Xilinx TRACE (Timing Analyzer) tool, and the number of control steps (C-steps) provided by Asserta. Notice that the area values are more than the initial task areas shown in Table 9.2. This is because the initial task area is simply the ALU area and does not include the interconnect and controller components.

The HLS tool can be used in a lighter form in order to obtain area and performance estimates on the RTL design. This light-weight version of HLS will be used by the partitioning tools to get quick design estimates. This would invoke only the initial design space exploration phases of HLS (refer to Figure 9.5). As a light-weight estimator, the HLS tool simply selects a *component bag* (a collection of RTL components) that corresponds to an efficient RTL implementation and also provides the corresponding estimates. The HLS estimator always over-estimates the design performance, ensuring that the actual HLS process will generate only a better RTL implementation. Also since the estimation process does not go through the entire (heavy-weight) HLS process, it will be considerably faster than the actual HLS.

9.4 Experimental Results

We developed two versions of the JPEG compression algorithm: In the **static**-JPEG version the board was configured only once to perform DCT . Whereas in the **dynamic**-JPEG version, we have generated multiple configurations for the DCT subtask using the SPARCS design tools. Each of these configurations were down-loaded once on the board to perform the DCT on an entire image file.

In order to obtain the one-time configuration of DCT for the static-JPEG version, we first partitioned DCT using the SPARCS spatial partitioner and used the Asserta synthesis tool [32] to synthesize the DCT partitions onto the FPGAs. The board having two XC4005 FPGAs, was then configured *once* to perform DCT, and the rest of the JPEG subtasks written in software ran on the host PC .

The static-JPEG codesign was tested on the six image files of varying sizes, listed in Table 9.4.

Table 9.4: **Execution times for Static-JPEG**

Images	No. of pixels	Static JPEG Codesign		
		JPEG	DCT in hardware	
		exec. (sec)	exec. (sec)	% of JPEG exec.
Scenery	592704	34.88	31.01	88.9
Portrait	576000	34.06	30.14	88.5
Parrots	294912	17.57	15.44	87.8
Turbo	69888	4.19	3.67	87.3
Group	56400	3.36	2.96	88.1
XV	54896	3.31	2.88	87

Table 9.5: **Execution times for Dynamic-JPEG**

Images	Dynamic Codesign Version of JPEG		
	JPEG	DCT in hardware	
	exec. (sec)	exec. (sec)	% of JPEG exec.
Scenery	6.16	2.29	37.2
Portrait	6.15	2.24	36.4
Parrots	3.52	1.39	39.5
Turbo	1.24	0.71	57.3
Group	1.08	0.67	62.0
XV	1.09	0.67	61.5

Table 9.4 shows the execution times for the entire static-JPEG codesign as well as the static DCT subtask. The images are listed in the decreasing order of their sizes. The execution times were measured using the commercial *Quantify* tool from Pure Soft Inc. On an average, DCT consumes about 88% of the total static-JPEG execution time, the rest of JPEG tasks running on the host PC consume about 6%, and the remaining time is spent in file I/O.

The DCT task graph shown in Figure 9.3 and the board architecture (two XC4005s and 8K 16-bit memories) were then fed to the SPARCS design system. The DCT task graph was taken through the design process as described in Section 9.3. The HLS tool in the SPARCS system produced a reconfiguration schedule and a collection of RTL designs. The reconfiguration schedule for the DCT task graph is shown in Figure 9.4. Using commercial synthesis tools, FPGA bit-map files were generated from the RTL designs. The configuration program was plugged into the JPEG software that ran on the host PC. The dynamic-JPEG codesign was then tested on the six image files. During the execution of the dynamic-JPEG codesign, the software running on the host PC would automatically down-load the DCT configurations one at a time, run the entire image file, and read the intermediate results back from the board. In this fashion, each of the DCT configurations was down-loaded only once for an entire image file.

Table 9.6: **Average Execution Times**

Images	# of 4x4 blocks	DCT Exec. time per 4x4 blocks in μsecs		Improv. factor
		Static	Dynamic	
Scenery	37044	837.1	61.82	13.54
Portrait	36000	837.3	62.22	13.46
Parrots	18432	837.7	75.41	11.11
Turbo	4368	839.3	162.5	5.16
Group	3525	839.7	190.1	4.42
XV	3431	839.9	195.3	4.30

Table 9.5 shows the execution times for the dynamic-JPEG codesign as well as the dynamically reconfigured DCT subtask. Notice that, on an average, DCT consumes only about 50% of the total dynamic-JPEG execution time, as opposed to 88% for the static-JPEG version. This shows that the dynamically reconfigured DCT version takes much less execution time when compared to the static DCT version.

We can take a closer look at the average time spent by both DCT versions on each 4x4 block of an image, as shown in Table 9.6. Notice that the dynamically reconfigured DCT version shows up to *13 times improvement* for the larger images. Also notice that, as the image size (or number of blocks) *decreases*, the average execution time for the dynamically reconfigured DCT *increases* and the improvement factor *decreases*. This is because *as the image size becomes smaller the reconfiguration overhead defeats the gain obtained due to dynamic reconfiguration.* This is the same reason why the percentage of the dynamic-JPEG execution time spent on the DCT subtask (shown in the last column of Table 9.5) increases as the image size decreases. In general, the minimum number of 4x4 blocks ($\mathcal{B}$) that is required for the dynamic-JPEG case to show an improvement over the static-JPEG case is given in Equation 9.17.

$$\mathcal{B} > \frac{(\mathcal{T} - 1) * \mathcal{R}}{\mathcal{D}_{static} - \mathcal{D}_{dynamic}} \tag{9.17}$$

where $\mathcal{B}$ is the number of blocks in an image, $\mathcal{D}_{static}$ and $\mathcal{D}_{dynamic}$ are the time taken (without the reconfiguration overhead) by the static and dynamic DCT versions to execute one image partition, $\mathcal{R}$ is the reconfiguration time for the FPGA board, and $\mathcal{T}$ is the number of temporal configurations generated for the dynamic JPEG .

We computed the values of $\mathcal{D}_{static}$ and $\mathcal{D}_{dynamic}$ to be 837.8 μsecs and 48.24 μsecs respectively. These values were computed by taking a product of the clock period and the number of clock cycles that the design requires to process one 4x4 block. Fifty temporal configurations ($\mathcal{T}$) were generated for DCT, and assuming a 10 milliseconds reconfiguration time, we can substitute these values in Equation 9.17 to get a value of 621 for $\mathcal{B}$. Therefore, there will be an improvement for any image that has more than 621 4x4 blocks. For images containing less than 621 blocks, the reconfiguration overhead becomes large enough to inhibit any gain due to dynamic reconfiguration.

9.5 Conclusions

In this chapter, we have presented an unified approach for design partition and synthesis onto dynamically reconfigurable multi-FPGA architectures. Using a real world design example, the DCT subtask of the JPEG still image compression standard, we showed that it is possible to achieve improvement if we perform dynamic reconfiguration instead of static (one-time) configuration. We also derived a general equation (Equation 9.17) and discussed the trade-off between the reconfiguration overhead and the gain achieved due to dynamic reconfiguration. We have presented a typical design flow through the SPARCS partitioning and synthesis environment, using the DCT as a case study example. The SPARCS design environment was used to automate the design process for dynamically reconfigurable architectures. The results presented in this chapter show that dynamic reconfiguration does provide a performance/cost advantage over static configuration for typical applications such as the JPEG algorithm that demand high performance.

Appendix A

BBIF Specification

The following sections provide details on the BBIF model. A formal and more detailed description of the BBIF model is available in [49].

A.1 BBIF Model and Formal Notations

A BBIF model can be represented as a four-tuple:

$$BBIF < Blocks,\ ControlDeps,\ IN_{ports},\ OUT_{ports} >$$

where *Blocks* is a set of behavior blocks, *ControlDeps* is a set of *control dependency* edges, where each edge $< B_i, B_j >$ represents the control flow from block B_i to B_j, and IN_{ports} *and* OUT_{ports} represent the set of design input and output ports respectively.

In the BBIF model the atomic storage element is a *carrier* represented as a tuple,

$$Carrier < Id,\ Width >$$

consisting of an index *Id* and a non-zero positive integer *Width*. The carrier *Id* is a unique index used for carrier set operations such as union and intersection. The design input and output ports are essentially carriers sets. A behavior block is an 8-tuple

$$BehaviorBlock < BlkId,\ Type,\ \mathcal{I},\ \mathcal{O},\ \mathcal{L},\ \mathcal{C},\ \mathcal{F},\ FG >$$

consisting of a block index (*BlkId*), a block type (*Type*), five carrier sets and a flow graph (*FG*). A behavior block can be either of type *compute* or *io*. Computations in a task are specified only within the *compute* blocks and interaction with the environment through the design ports are specified only within *io* blocks. The five carrier sets are:

- The set $\mathcal{I}(B_i)$ represents the set of *input* carriers of block B_i. These are input carriers that are passed from every parent block that branches to block B_i.
- The set $\mathcal{O}(B_i)$ represents the set of *output* carriers of block B_i. These are output carriers that are passed through the branches to every child of block B_i.
- The set $\mathcal{L}(B_i)$ represents the set of *local* carriers of block B_i. These carriers are visible only within block B_i and are used to capture the data flow across computations within the block.

- The set $\mathcal{C}(B_i)$ represents the set of *constants* that are visible only within block B_i. A constant is essentially a carrier with an additional *string* field that represents the actual constant value.
- The set $\mathcal{F}(B_i)$ represents the set of *flag* carriers of block B_i. These carriers are visible only within block B_i and are used to hold the resulting values of conditional expressions in the behavior. The flags are used for conditional branching at the end of the block.

In addition, a behavior block also consists of a *flow graph* FG, represented as a tuple,

$$FG < OprNodes,\ DataFlowDeps >$$

consisting of *operation nodes* ($OprNodes$) and *data dependency* edges ($DataFlowDeps$). An operation node is a 5-tuple

$$Operation < OprId,\ OprType,\ Inputs,\ Outputs,\ ConDeps >$$

consisting of a unique operation index ($OprId$), the operation type ($OprType$), the *input* carrier set, the *output* carrier set and an explicit control dependency set ($ConDeps$). Each operation O_i, has a set of input carriers $Inputs(O_i)$ that are *read* and a set of output carriers $Outputs(O_i)$ that are *written*. Both these sets may contain zero or more carriers. A data *flow dependency* is a directed edge between a parent operation O_i and a child operation O_j, represented as a tuple $< O_i,\ O_j >$. This dependency edge exists if and only if the following condition is satisfied:

$$DataFlowDependency < O_i,\ O_j > \iff (Outputs(O_i) \cap Inputs(O_j)) \neq \emptyset$$

The operation nodes in a block follow *single assignment semantics* by writing exactly once to a particular carrier. In other words, any *output, local, or flag* carrier in a block will appear exactly in only one *output* carrier set of an operation node. Therefore, there are no *anti* or *output* dependencies and the order of operations in the BBIF specification does not matter in deriving the flow graph.

The operations in the BBIF are classified as *pre-defined* and *user-defined*. There are three pre-defined operation types in the BBIF. The *io* block supports the two types namely, `getport` and `putport` to facilitate design port accesses. The third pre-defined operation type is the `transfer` that is supported in any behavior block. The transfer operation denotes an assignment of one carrier to another, and corresponds to an assignment statement in the behavior. The user-defined operations are uninterpreted. In other words, the synthesis system does not attach any functional semantics to the user-defined operations and expects the user to specify a component library that supports these operations.

A.2 Translation and Profiling

Figure A.1 shows the VHDL specification of an ALU example. The ALU takes two data inputs and a mode of operation, and generates a result. Depending on the mode of operation the ALU generates the *sum*, *difference*, *product*, or the *sum of squares* of the inputs. Figure A.2 shows the BBIF that was automatically translated from the VHDL specification of the ALU. The start *io* block `Blk_2` reads the design ports into the corresponding carriers and passes them to `Blk_3`. The TRUE_BRANCH statement represents an unconditional branch to a subsequent block. `Blk_3`

```
entity ALU is  port ( Data1, Data2 : in integer;
         Mode : in bit_vector(1 downto 0);
         RESULT : out integer
       );
end ALU;

architecture behavior of ALU is
begin
  compute: process
    variable A, B, value : integer;
    variable M : bit_vector(1 downto 0);
  begin
    A := Data1;
    B := Data2;
    M := Mode;
    case M is
      when "00" => value := A + B;
      when "01" => value := A - B;
      when "10" => value := A * B;
      when others => value := (A * A) + (B * B);
    end case;
    RESULT <= value;
  end process;
end behavior;
```

Figure A.1: VHDL Specification of an ALU

```
(INPORT (data1 16) (data2 16) (mode 2))
(OUTPORT (result 16))

(BB Blk_2
    (LOCAL (a 16) (b 16) (m 2))
1  (GET_PORT (data1) (a)) ()
2  (GET_PORT (data2) (b)) ()
3  (GET_PORT (mode) (m)) ()
(TRUE_BRANCH Blk_3(a b m))
)
(BB Blk_3 ( (a 16) (b 16) (m 2) )
    (LOCAL (flag_1 1) (flag_2 1) (flag_3 1))
    (CONSTANT (c12 2 00) (c15 2 01) (c18 2 10))
4  (eq (m c12) (flag_1)) ()
5  (eq (m c15) (flag_2)) ()
6  (eq (m c18) (flag_3)) ()
(flag_1 Blk_4(a b)
(flag_2 Blk_5(a b)
(flag_3 Blk_6(a b) Blk_7(a b))))
)
(BB Blk_4 ( (a 16) (b 16) )
    (LOCAL (value 16))
7  (plus (a b) (value)) ()
(TRUE_BRANCH Blk_8(value))
)
⋮
(BB Blk_6 ( (a 16) (b 16) )
    (LOCAL (value 16))
9  (mult (a b) (value)) ()
(TRUE_BRANCH Blk_8(value))
)
(BB Blk_7 ( (a 16) (b 16) )
    (LOCAL (t22 32) (t23 32) (value 16))
10  (mult (a a) (t22)) ()
11  (mult (b b) (t23)) ()
12  (plus (t22 t23) (value)) ()
(TRUE_BRANCH Blk_8(value))
)
(BB Blk_8 ( (value 16) )
13  (PUT_PORT (value result) ()) ()
(TRUE_BRANCH Blk_2())
)
```

Figure A.2: BBIF Specification of the ALU Example

has three inputs `a`, `b` and `m`, whereas `Blk_2` does not have any. The *compute* block `Blk_3` performs all the condition evaluations of the case statement and generates three flags. Based on the values of these flags, four branches arise from `Blk_3` leading to the blocks `Blk_4`,`Blk_5`, `Blk_6` and `Blk_7`. The four blocks perform the four types of the ALU operations. For example, the sum of squares is performed in the three operation statements of `Blk_7`. Each of these four blocks call the *io* block `Blk_8` to write the results to the output port. Note that `Blk_8` calls the start block `Blk_2` forming an overall infinite loop that represents the implicit loop of corresponding VHDL process.

A.3 Component Library and Functional Unit Instantiation

The component library (C_{lib}), supplied by the user, specifies a list of combinational and sequential components with a list of operations supported by them. For every *user-defined* operation in the BBIF, there should exist at least one component in the library that supports that operation. For sequential components and for components that can support multiple operations, the user is also expected to provide the control signal that facilitates the selection of each of these operations. This information will be used by the synthesis system while generating the control logic.

Figure A.3 shows a portion of a typical component library. The class of a component denotes whether it is combinational (denoted by ALU) or sequential (denoted by REG). The first component `compare`, supports multiple operation types as specified in its MODE field, and its CONTROL field provides the control signal information for each operation type it supports. Components can be parameterized over their port sizes as well as over the number of ports. In the figure, the component that supports the bit-wise `and` operation is parameterized both on the number of inputs and port widths. Since all user-specified operations in the input description are uninterpreted, the library should provide all relevant information that the synthesis process might subsequently require. The SIGNATURE field specifies the ports of the component that are used to support an operation type. For example, `compare` has two inports and three outports while one of its operation type, `grt` uses the first two inports to read inputs and uses the third outport to write the output.

Given a BBIF specification and a component library, the HLS system performs *resource set generation*. For each unique operation type in the BBIF, one or more *functional units* are generated from the parameterized components in the given library. A functional unit is a library component that is instantiated with specific values to its generic parameters. This is done by matching the type of each BBIF operation with the MODE field of each component. Consequently, from the input and output carrier sets of the BBIF operation the generic parameters of the component are instantiated, resulting in a new functional unit. For example, the `eq` operations 4, 5 and 6 of block `Blk_3` in Figure A.2 would lead to a functional unit instantiation from component `compare` with generic parameter values of `width1` = 2, and `width2` = 1. The functional units are unique with respect to the component name and the parameter values. If *resource folding* needs to be performed, functional units may remain unique only based on the component name.

```
⋮
(COMP compare (width1 width2)
  (CLASS ALU)
  (MODE less grt eq)
  (INPORT (a width1) (b width1))
  (OUTPORT (c width2) (d width2) (e width2))
  (SIGNATURE
   (less (1 2) (1))
   (grt (1 2) (2))
   (eq (1 2) (3))
  )
  (CONTROL 2 (NOP 00) (less 01) (grt 10) (eq 11))
)

(COMP and (ins width1 width2)
  (CLASS ALU)
  (MODE and)
  (INPORT (ins width1))
  (OUTPORT (out width2))
  (CONTROL)
)

(COMP REG (bitwidth)
  (CLASS REGISTER)
  (MODE reg)
  (INPORT (input bitwidth))
  (OUTPORT (output bitwidth))
  (CONTROL 2 (NOP 00) (LOAD 01) (RESET 10))
)
⋮
```

Figure A.3: Snapshot of a Typical Component Library

Bibliography

[1] A. A. Duncan, D. C. Hendry and P. Gray. "An Overview of the Cobra-ABS High-Level Synthesis System for Multi-FPGA Systems". In *Proceedings of FPGAs for Custom Computing Machines (FCCM)*, pages 106–115, Napa Valley, California, 1998.

[2] A.E. Casavant, D.D. Gajski, and D.J. Kuck. "Automatic Design with Dependence Graph". In *17th Design Automation Conference"*, pages 506–515, 1980.

[3] B. Kernighan, D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1978.

[4] B. S. Baker, D. J. Brown, and H. P Katseff. A 5/4 Algorithm for Two-Dimensional Packing. *Journal of Algorithms*, 2:348–368, 1981.

[5] B. S. Baker, E. G. Coffman, and R. L Rivest. Orthogonal Packings in Two Dimensions. *Siam Journal of Computing*, 9:846–855, November 1980.

[6] B. S. Baker and J. S. Schwarz. Shelf Algorithms for Two-Dimensional Packing Problems. *Siam Journal of Computing*, 12:508–525, August 1983.

[7] C.M. Fiduccia, R.M. Mattheyses. "A Linear Time Heuristic for Improving Network Partitions". In *Proc. of 19th Design Automation Conference*, pages 175–181, 1982.

[8] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance Bounds For Level-Oriented Two-Dimensional Packing Algorithms. *Siam Journal of Computing*, 9:808–826, November 1980.

[9] D. D. Gajski, F. Vahid, et al. . "Specification and Design of Embedded Systems". In *Prentice-Hall Inc.*, Upper Saddle River, NJ, 1994.

[10] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *"High-Level Synthesis, Introduction to Chip and System Design"*. Kluwer Academic Publishers, 1992.

[11] D. Huang, A.B. Kahng. "Multi-Way System Partitioning into a Single Type or Multiple Types of FPGAs". In *Proc. of 3rd Int. Symp. FPGAs*, pages 140–145, 1995.

[12] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA,, 1989.

[13] D.E. Thomas J.K. Adams, H. Schmit. "A Model and Methodology for Hardware-Software Codesign". In *IEEE. Design & Test of Computers*, pages 6–15, September 1992.

[14] E.D.Lagnese and D.E.Thomas. "Architerctural partitioning of system level synthesis of integrated circuits". In *IEEE Transactions on CAD*, volume 9, No.9, pages 847–860, July 1991.

[15] J. M. Emmert and D. K. Bhatia. *TABU Search for Fast Timing Driven Placement of Circuits on FPGAs.* University of Cincinnati Technical Report Number: TR219/09/98/ECECS, 1998.

[16] J. M. Emmert and D. K. Bhatia. A Methodology for Fast FPGA Floorplanning. In *ACM Seventh International Symposium on Field-Programmable Gate Arrays*, pages 47–56, Feburary 1999.

[17] J. M. Emmert, A. Randhar, and D. K. Bhatia. Fast Floorplanning for FPGAs. In *Lecture Notes in Computer Science*, volume 1482, pages 129–138. Springer-Verlag, 1998.

[18] D.E. Thomas et al. *"Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench".* Kluwer Academic Publishers, 1990.

[19] F. Vahid. "Functional Partitioning Improvements Over Structural Partitioning for Packaging Constraints and Synthesis: Tool Performance". In *ACM Transactions on Design Automation of Electronic Systems, Vol 3, No. 2*, pages 181–208, April 1998.

[20] F. Vahid. "Techniques for Minimizing and Balancing I/O During Functional Partitioning". In *IEEE Trans. on CAD, vol. 18 No. 1*, pages 69–75, Jan 1999.

[21] F. Vahid, D.D. Gajski. "Specification Partitioning for System Design". In *Proc. of 29th Design Automation Conference*, pages 219–224, 1992.

[22] F. Vahid, D.D. Gajski. "Incremental Hardware Estimation During Hardware/Software Functional Partitioning". In *IEEE Trans. on VLSI Systems, Vol 3, No 3*, September 1995.

[23] G. De Micheli. "Computer-Aided Hardware Software Codesign". In *IEEE Micro*, pages 10–16, Aug 1994.

[24] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

[25] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul and R. Vemuri. "A Unified Specification Model of Concurrency and Coordination for Synthesis from VHDL". In *Proceedings of the 4th International Conference on Information Systems Analysis and Synthesis (ISAS)*, July 1998.

[26] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, R. Vemuri. "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures". In *Proceedings of Parallel and Distributed Processing, (RAW98)*, pages 31–36. Springer, March 1998.

[27] IEEE Standards Office. "IEEE Standard VHDL Language Reference Manual". In *IEEE Standards Office*, New York, NY, 1993.

[28] Altera Inc. *http://www.altera.com.*

[29] Xilinx Inc. *http://www.xilinx.com.*

[30] J. Henkel, R. Ernst. "The Interplay of Run-time Estimation and Granularity in HW/SW Partitioning". In *Fourth International Workshop on Hardware/Software codesign*, pages 52–58, March 1996.

[31] J. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press, 1975.

[32] J. Roy, N. Kumar, R. Dutta and R. Vemuri. "DSS: A Distributed High-Level Synthesis System". In *IEEE Design and Test of Computers*, June 1992.

[33] J.Hou, W. Wolf. "Process Partitioning for Distributed Embedded Systems". In *Fourth International workshop on Hardware/Software codesign*, pages 70–76, March 1996.

[34] K. Kucukcakar, and A. Parker. "CHOP: A constraint-driven system-level partitioner". In *Proceedings of the Conference on Design Automation*, pages 514–519, 1991.

[35] K. Roy-Neogi, C. Sechen. "Multiple FPGA Partitioning with Performance Optimization". In *Proc. of 3rd Int. Symp. FPGAs*, pages 146–151, 1995.

[36] M. Kaul and R. Vemuri. "Temporal Partitioning combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs". In *Design, Automation and Test in Europe, DATE*, pages 202–209. IEEE Computer Society Press, 1999.

[37] K. Kucukcakar. *System-Level Synthesis Techniques With Emphasis on Partitioning and Design Planning.* PhD thesis, University of Southern California, CA, 1991.

[38] N. Kumar. *High Level VLSI Synthesis for Multichip Designs.* PhD thesis, University of Cincinnati, 1994.

[39] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand, Reinhold, NY, 1989.

[40] K. Li and K. Cheng. On Three-Dimensional Packing. *Siam Journal of Computing*, 19:847–867, October 1990.

[41] M. J. Farland. *"Value Trace".* Carnegie Mellon University, Internal Report, Pittsburgh, PA, 1978.

[42] M. Vootukuru. "Partitioning of Register Transfer Level Designs for Multi-FPGA Synthesis". In *VIUF Conference*, Spring 1996.

[43] M. Vootukuru. "Performance Estimation and Partitioning of VHDL Models for FPGA Implementation". Master's thesis, University of Cincinnati, USA, July 1996.

[44] Min Xu, F.J. Kurdahi. "Layout-Driven High Level Synthesis for FPGA Based Architectures". In *Proc. of Design Automation and Test in Europe*, pages 446–450, February 1998.

[45] M.J.McFarland and T.Kowalski. "Incorporating bottom-up design into hardware synthesis". In *IEEE Transactions on CAD*, volume 9, No.9, September 1990.

[46] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-Packing-Based Module Placement. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 472–479, November 1995.

[47] N. A. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, Boston, 1993.

[48] N. Kumar, V. Srinivasan, and R. Vemuri. "Hierarchical Behavioral Partitioning for Multi Component Synthesis". In *Proc. European Design Automation Conference*, pages 212–219, 1996.

[49] N. Narasimhan. *"Formal-Assertions Based Verification in a High-Level Synthesis System"*. PhD thesis, University of Cincinnati, ECECS Department, 1998.

[50] N. Narasimhan, et al. Theorem Proving Guided Discovery of Formal Assertions in Resource-Constrained Scheduler for High-Level Synthesis. In *Intl. Conference on Computer Design*, Oct 1998.

[51] N-S Woo, J. Kim. "An Efficient Method of Partitioning Circuits for Multi-FPGA Implementations". In *Proc. 30th ACM/IEEE Design Automation Conference*, pages 202–207, 1993.

[52] N. Woo, A.E. Dunlop, W. Wolf. "Codesign from Cospecification". In *IEEE. Computer*, pages 42–47, January 1994.

[53] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. In *Design Automation for Embedded Systems, 2, Kluwer Academic Publishers*, pages 5–32, 1997.

[54] P. Harper, S. Krolikoski, and O. Levia. "Using VHDL as a Synthesis Language in the Honeywell VSYNTH System". In *Proceedings of Computer Hardware Description Languages and their Applications*, pages 315–330. Elsevier, June 1989.

[55] P. Sawkar, D. Thomas. "Multi-way Partitioning for Minimum Delay for Look-Up Table Based FPGAs". In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 201–205, 1995.

[56] P.G.Paulin and J.P.Knight. "Force Directed Scheduling for the Behavior Synthesis of ASICs". In *IEEE Transactions on CAD*, volume 8, pages 661–679, June 1989.

[57] P.K. Chan, M. Schlag, J. Zien. "Spectral-Based Multi-Way FPGA Partitioning". In *Proc. of 3rd Int. Symp. FPGAs*, pages 133–139, 1995.

[58] R. Ernst, J. Henkel, T. Benner. "Hardware-Software Cosynthesis for Microcontrollers". In *IEEE. Design & Test of Computers*, pages 64–75, December 1993.

[59] R. K. Gupta and G. De Micheli. "Partitioning of funtional models of synchronous digital systems". In *Proceesings of the International Conference on Computer-Aided Design*, pages 216–219, 1990.

[60] R. Kuznar, F. Brglez, B. Zajc. "Multi-way Netlist Partitioning into Heterogeneous FPGAs and Minimization of Total Device Cost and Interconnect". In *Proc. 31st ACM/IEEE Design Automation Conference*, pages 228–243, 1994.

[61] R. Vemuri. *Genetic algorithms for Partitioning, Placement, and Layer Assignment for Multi-chip Modules.* PhD thesis, University of Cincinnati, USA, July 1994.

[62] R. Vemuri, H. Carter, and P. Alexander. "Board and MCM Level Synthesis for Embedded Systems: The COMET Cosynthesis Environment". In *Proceedings of First Annual RASSP Conference*, August 1994.

[63] R.Camposano and V.J. Eijndhoven. "Partitioning a design in structural synthesis". In *Proceedings of the European Conference on Design Automation*, pages 14–18, 1987.

[64] R.K. Gupta, G.De Micheli. "Hardware-Software Cosynthesis for Digital Systems". In *IEEE. Design & Test of Computers*, pages 29–40, September 1992.

[65] R.K. Gupta, G.De Micheli. "System-level Synthesis using Re-programmable Components". In *Proc. European Design Automation Conference*, pages 2–7, 1992.

[66] S. Govindarajan and R. Vemuri. "Cone-Based Clustering Heuristic for List Scheduling Algorithms". In *Proceedings of European Design & Test Conference (ED&TC)*, pages 456–462, Paris, France, March 1997. IEEE Computer Society. ISBN 0-8186-7786-4.

[67] S. Govindarajan and R. Vemuri. "An Efficient Clustering-Based Heuristic for Time-Constrained Static-List Scheduling. In *Proceedings of the IEEE Design, Automation and Test in Europe, DATE Conference*, 2000.

[68] S. Govindarajan, I. Ouaiss, V. Srinivasan, M. Kaul and R. Vemuri. "An Effective Design System for Dynamically Reconfigurable Architectures". In *Proceedings of Sixth Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 312–313, Napa, California, April 1998. IEEE Computer Society. ISBN 0-8186-8900-5.

[69] S. Govindarajan, V. Srinivasan, P. Lakshmikanthan, and R. Vemuri. A Technique for Dynamic High-Level Exploration During Behavioral-Partitioning for Multi-Device Architectures. In *Proc. of the 13th IEEE Intl. Conf. on VLSI Design*, Calcutta, India, January 2000. Received the Best Paper Award.

[70] S. Hauck, G. Borriello. "Logic Partition Orderings for Multi-FPGA Systems". In *Proc. of 3rd Int. Symp. FPGAs*, pages 32–38, 1995.

[71] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi. "Optimization by Simulated Annealing". In *Science, vol 220, no.4598,*, pages 671–680, 1983.

[72] S. P. Levitan et al. "Using VHDL as a Language for Synthesis of CMOS VLSI Circuits". In *Proceedings of Computer Hardware Description Languages and their Applications*, pages 331–346. Elsevier, June 1989.

[73] S. M. Sait and H. Youssef. *VLSI Physical Design Automation.* IEEE Press, 1995.

[74] S.Govindarajan and R.Vemuri. Dynamic Bounding of Successor Force Computations in the Force Directed List Scheduling Algorithm. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 752–757, Austin, Texas, October 1997.

[75] U. Steinhausen, R. Camposano, et al. "System-Synthesis Using Hardware / Software Codesign". In *International Workshop on Hardware-Software Co-Design*, October 1993.

[76] V. Catania, M. Malgeri, M. Russo. "Applying Fuzzy Logic To Codesign Partitioning". In *IEEE Micro*, pages 62–70, June 1997.

[77] V. Srinivasan. *Partitioning for FPGA-Based Reconfigurable Computers.* PhD thesis, University of Cincinnati, USA, August 1999.

[78] V. Srinivasan, R. Vemuri. "Task-level Partitioning and RTL Design Space Exploration for Multi-FPGA Architectures". In *Int. Symposium on Field-Programmable Custom Computing Machines*, April 1999.

[79] V. Srinivasan, Ram Vemuri, Ranga Vemuri. Genetic Algorithms for Physical Design of Multi-Chip Modules. *Submitted to the IEEE Trans. on VLSI Systems.*

[80] V. Srinivasan, S. Radhakrishnan, and R. Vemuri. Hardware/Software Partitioning with Integrated Hardware Design Space Exploration. In *Proc. of Design Automation and Test in Europe*, pages 28–35, February 1998.

[81] Vinoo Srinivasan. *"Partitioning in Reconfigurable Computing Environments"*. PhD thesis, University of Cincinnati, ECECS Department, 1999.

[82] W-J Fang, A. Wu. "A Hierarchical Functional Structuring and Partitioning Approach for Multi-FPGA Implementations". In *IEEE Trans. on CAD, vol. 9 No. 5*, pages 500–511, Nov. 1990.

[83] W. J. Fang and A. C. H. Wu. "Integrating HDL Synthesis and Partitioning for Multi-FPGA Designs". In *IEEE Design and Test of Computers*, pages 65–72, April-June 1998.

[84] "Wildforce". Wildforce Reference Manual, Document #11849-0000.

[85] M. Xu. *Linking High Level Synthesis with Physical Design.* PhD thesis, University of California, Irvine, 1997.

[86] Y. Chen, Y. Hsu, and C. King. "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures". In *IEEE Transactions on VLSI systems*, volume 2, No. 1, pages 21–32, March 1994.

[87] T. Yamanouchi, K. Tamakashi, and T. Kambe. Hybrid Floorplanning Based on Partial Clustering and Module Restructuring. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 478–483, 1996.

[88] Atmel Inc., "Configurable Logic: Design and Application Book",http://www.atmel.com.

[89] B. L. Hutchings and M. J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications", *Field-Programmable Logic and Applications, FPL 1995, pp. 419-428.*

[90] M. Dorfel and R. Hofmann, "A Prototyping System for High Performance Communication Systems", *IEEE Workshop on Rapid System Prototyping, RSP 1998, pp. 84-88.*

[91] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki and A. Agarwal, "Logic emulation with virtual wires", *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, v16, n6, June 1997.*

[92] R. D. Hudson, D. I. Lehn and P. M. Athanas, "A Run-Time Reconfigurable Engine for Image Interpolation", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1998, pp. 88-95.*

[93] M. J. Wirthlin and B. L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 1996, pp. 122-128.*

[94] M. J. Wirthlin and B. L. Hutchings, "A Dynamic Instruction Set Computer", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1995, pp. 99-106.*

[95] K. M. GajjalaPurna and D. Bhatia, "Emulating Large Designs on Small Reconfigurable Hardware", *IEEE Workshop on Rapid System Prototyping, RSP 1998, pp. 58-63.*

[96] M. B. Gokhale and J. M. Stone, "NAPA C:Compiling for Hybrid RISC/FPGA Architectures", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1998, pp. 126-135.*

[97] W. Luk, N. Shirazi and P. Cheung, "Automating Production of Run-Time Reconfigurable Designs", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1998, pp. 147-156.*

[98] M. Chu, N. Weaver, K. Sulimma, A. DeHon and J. Wawrzynek, "Object Oriented Circuit-Generators in Java", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1998, pp. 158-166.*

[99] J. Spillane and H. Owen, "Temporal Partitioning for Partially-Reconfigurable-Field-Programmable Gate", *Reconfigurable Architectures Workshop in 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, IPPS/SPDP 1998, pp. 37-42.*

[100] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul and R. Vemuri, "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures", *Reconfigurable Architectures Workshop in 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, IPPS/SPDP 1998, pp. 31-36.*

[101] K. Roy-Neogi and C. Sechen, "Multiple FPGA partitioning with performance optimization", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 1995, pp. 146-152.*

[102] P. Chan, M. Schlag and J. Zien, "Spectral-based multi-way FPGA partitioning", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 1995, pp. 133-139.*

[103] W. Fang and A. Wu, "A Hierarchical Functional Structuring and Partitioning Approach for Multiple-FPGA Implementations", *IEEE Transactions on Computer-Aided Design, v16, n10, Oct 1997, pp. 1188-1195.*

[104] M. Kaul and R. Vemuri, "Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures", *Design and Test in Europe, DATE 1998, pp. 389-396.*

[105] M. Kaul, R. Vemuri, S. Govindarajan and I. Ouaiss, "An Automated Temporal Partitioning and Loop Fission approach for FPGA based reconfigurable synthesis of DSP applications", *36th Design Automation Conference, DAC 1999.*

[106] C. H. Gebotys, "Optimal Synthesis of Multichip Architectures", *IEEE ICCAD, Nov. 1992, pp. 238-241.*

[107] S. Trimberger, "Scheduling designs into a Time-Multiplexed FPGA", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 1998, pp. 153-160.*

[108] R. Niemann and P. Marwedel, "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming", *Proceedings of the European Design and Test Conference, ED&TC, 1996.*

[109] A. Kalavade, "System-Level Codesign of Mixed Hardware-Software Systems", Ph.D. Dissertation, University of California, Berkeley, 1995.

[110] D. S. Rao and F. Kurdahi, "Hierarchical Design Space Exploration for a Class of Digital Systems", *IEEE Transactions on VLSI, v 1, n 3, Sept 1993, pp. 282-294.*

[111] H. Schmit, L. Arnstein, D. Thomas and E. Lagnese,"Behavioral Synthesis for FPGA-based Computing", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1994, pp. 125-132.*

[112] R. Dutta, J. Roy, and R. Vemuri, "Distributed Design Space Exploration for High-Level Synthesis Systems", *29th Design Automation Conference, DAC 1992, pp. 644-650.*

[113] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[114] M. Wolf, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishers, 1996.

[115] S. Y. Kung, *VLSI Array Processors*, Prentice Hall 1988.

[116] S. Trimberger, "A Time-Multiplexed FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1997, pp. 22-28.*

[117] S.M. Scalera, J. R. Vazquez,"The Design and Implementation of a Context Switching FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM 1998, pp. 78-85.*

[118] WILDFORCE Reference Manual, *Document #1189 - Release Notes, Annapolis Micro Systems, Inc..*

[119] Y. Hung, A. C. Parker, "High-Level Synthesis with Pin Constraints for Multiple-Chip Designs", *29th Design Automation Conference, 1992.*

[120] G. K. Wallace, "The JPEG Still Picture Compression Standard", *ACM Communications, 1991.*

[121] P. Hansen, B. Jaumard and V. Mathon, "Constrained Nonlinear 0-1 programming", *ORSA Journal of Computing, v5, n2, 1993, pp.97-119.*

[122] F. Glover and E. Woolsey, "Converting the 0-1 Polynomial Programming Problem to a 0-1 Linear Program", *Operations Research 21:1, 1974, pp. 156-161.*

[123] Pierre G. Paulin and John P. Knight, "Force Directed Scheduling for the behavioral synthesis of ASICs," IEEE Trans. Computer Aided Design, Vol.8, pp. 661-679, June 1989.

[124] Raul Camposano, Wayne Wolf, "High-Level VLSI Synthesis", Kluwer Academic Publishers, 1991.

[125] Daniel Gajski, Nikil Dutt, "High-Level Synthesis", Kluwer Academic Publishers, 1992.

[126] D.D. Gajski, N.D. Dutt, and B.M. Pangrle, "Silicon compilation (tutorial)," in Proc. IEEE 1986 Custom Integrated Conf. (Rochester NY), May 1986, pp. 102-110.

[127] Jan Vanhoof et. al., "High-Level Synthesis for Real-Time Digital Signal Processing", Kluwer Academic Publishers, 1993.

[128] J. Lee, Y. Hsu, and Y. Lin, "A new Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis," Proc. of the Int. conf. on Computer-Aided Design, pp. 20-23, 1989.

[129] I-C. Park and C-M. Kyung, "Fast and Near Optimal Scheduling in Automatic Data Path Synthesis," Proc. of the 28th DAC, pp. 680-685, 1991.

[130] R. Camposano, "Path-Based Scheduling for Synthesis," IEEE Trans. on CAD of Integ. Cir. and Systems, vol. 10, no. 1, pp. 85-93, Jan 1991.

[131] Sriram Govindarajan and Ranga Vemuri, "Cone-Based Clustering Heuristic for List-Scheduling Algorithms", Proceedings of the ED&TC 1997, Session 9C: New Ideas in Scheduling.

[132] W.J.F Verhaegh, et al., "Improved Force-Directed Scheduling", Proceedings of the EDAC, pp. 430-435, 1991.

[133] W.J.F Verhaegh, et al., "Efficiency Improvements for Force-Directed Scheduling", Proceedings of the ICCAD, pp. 286-291, 1992.

[134] S.Davidson et.al., "Some Experiments in local microcode compaction for horizontal machines", IEEE Trans. Comp., pp. 460-477, July 1981.

[135] Phillip E. Mattison, "Practical Digital Video with Programming Examples in C", John Wiley & Sons, Inc., 1994.

[136] S.Y.Kung, H.J.Whitehouse, T.Kailath, "VLSI and Modern Signal Processing", Prentice-Hall, Inc., 1985.

[137] Michael Wolfe, "High Performance Compilers for Parallel Computing", Addison-Wesley Pub., 1996.

[138] Jacek M. Zurada, "Introduction to Artificial Neural Systems", West Publishing Company, 1992.

[139] M. Vasilko and D. Ait-Boudaoud, "Architectural Synthesis Techniques for Dynamically Reconfigurable Logic", *FPL'96.*

[140] Iyad Ouaiss et al., "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures", *Fifth Reconfigurable Architectures Worshop*, March 1998.

[141] Gregory K. Wallace, "The JPEG Still Picture Compression Standard", *Communications of the ACM*, pages 30-44, April 1991.

[142] Mattison E. Phillip, "Practical Digital Video with Programming in C", *Wiley, New York*, 1994

[143] Naren Narasimhan et al., "Rapid Prototyping of Reconfigurable Coprocessors", *International Conference on Application-specific Systems, Architectures and Processors*, August 1996.

[144] M. Kaul and R. Vemuri, "Optimal Temporal Parititioning and Synthesis for Reconfigurable Architectures", to appear in *Design and Test in Europe '98.*

[145] C.H. Gebotys and M. I. Elmasry, "Optimal VLSI Architectural Synthesis: Area, Performance and Testability", *Kluwer Academic Publishers, 1992.*

[146] B. Landwehr, P. Marwedel and R. Domer, "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming", *Proceedings of the EuroDac, p90-95, 1994.*

[147] M. Vootukuru, R. Vemuri, and N. Kumar, "Resource Constrained RTL Partitioning for Synthesis of Multi-FPGA Designs", *Proceedings of the 10th International Conference on VLSI Design, IEEE Press, 12 pages, 140-144, January 1997.*

[148] Cohoon J. and W. Paris, "Genetic Placement", *IEEE Trans. on CAD, vol. CAD-6 No. 6, pages 956–964, November 1987.*

[149] Shahookar K. and P. Mazumdar, "A Genetic Approach to Standard Cell Placement using Meta-genetic Parameter Optimization", *IEEE Trans. on CAD, vol. 9 No. 5, pages 500–511, Nov. 1990.*

[150] S. Raman and L. M. Patnaik, "Performance-Driven MCM Partitioning Through an Adaptive Genetic Algorithm", *IEEE Trans. on VLSI Systems, vol. 4(4), pp. 434-444, Dec 1996.*

[151] Sreenivasa Rao D., F.J.Kurdahi, "Hierarchical design space exploration for a class of digital systems", *IEEE Transactions on VLSI Systems, 1993.*

[152] Jan Vanhoof et al., "High-Level Synthesis for Real-Time Digital System Processing", *Kluwer Academic Publishers, 1993.*